Alberto Paoluzzi and Giorgio Scorzelli

BIM geometry with Julia Plasm

Functional language for CAD programming

May 28, 2024

Springer Nature

1	3	
Pre/Post conds	4	→ 5
Example	6	

Contents

Part I
Basic Concepts

Chapter 1
Introduction to Julia Programming

Many programming languages are developed for writing computer programs. Let's imagine their set covered, in a mathematical sense, by a few subsets called *programming paradigms* that characterize the *programming style*, i.e., how the computation is performed, how data are transformed or stored, and other aspects of calculation. Some languages follow a single paradigm; other languages are multiparadigm since their programs may be written adopting more than one style. Examples are Smalltalk (object-oriented), Haskell or ML (functional), Prolog (declarative). C++, Java, Python, and Julia are multiparadigm. Julia was designed for scientific programming, which currently leads for performance, simplicity, and expressiveness. Julia solved the *two-language problem* [**?**], so unifying program prototyping and optimization. This book begins with a chapter introducing the Julia multi-paradigm which blends high description power and simplicity with very fast computing ability. The chapter is not a programming primer, but is oriented to readers already able to code in other languages.

1.1 Basic syntax and type system

Julia combines features of productivity languages, such as Python or MATLAB, with characteristics of performance-oriented languages, such as C++ or Fortran. Syntactically, Julia is easy and fast to write and debug, and also enjoys a great collection of packages.

Basic syntax

Julia strongly resembles Python for generic code, and MATLAB for algebraic calculus with matrices and vectors. Only a few syntax notions are actually necessary to allow the reader to understand the scripts in this book. A set of differences between the languages Julia, Python, and R, full of scriptlets, may be found in https://cheatsheets.quantecon.org, suggested to readers.

The comment *lines* start with a character `#`. All text following it on the line is considered a comment and skipped by the compiler. *Multiline* comments, called block-comments, start with `#=` and terminate with a reverse pair `=#`.

Even if actually multi-paradigm, Julia can be properly considered a functional language, as everything in Julia is an *expression* generating a *value*.

There are several basic types of numbers: e.g., `Int64`, `Float64`, `Complex`{`Float64`}, `Rational`{`Int64`}. `String` values are created with "<string chars>"; `Char` literals are written as `'a','b'`, etc. All arithmetic infix operators are available, as well as bitwise operators, Boolean values and operations, and primitive comparison operators.

Variables

Variables are declared while assigning a value to them. Accessing a non-previously declared variable is an error. *Names* of variables start with a *letter* or *underscore* and may have any number of alphanumeric characters, including `_` and `!`. It is also possible to use many Unicode characters, like π or $\in$, using their LaTeX names followed by the Tab key, like `\pi<Tab> # =>` π.

Values have a type, but variables do not so that a variable can be reassigned with values of the same or different types.

Naming conventions

1. In long names word separation can be indicated by underscores, but their use is discouraged unless the name would be hard to read otherwise;
2. names of `Type`s begin with a capital letter, and word separation is shown with *UpperCamelCase* instead of underscores: `AbstractFloat`;
3. names of *functions* and macros are in *lower case*, without underscores;
4. functions that modify their input have names that end in `!`. These functions are called *mutating* functions or *in-place* functions because they mutate their input. For large data structures, this one is rarely a good idea.

Control flow

Julia provides a variety of control flow constructs:

1. *Compound Expressions*: block **`begin...end`**, and **`;`** for expressions in sequence.
2. *Conditional Evaluation*: **`if...elseif...else...end`**, and **`<>?<>:<>`** (ternary operator).

3. *Short-Circuit Evaluation*: logical operators `&&` ("and"), and ‖ ("or"), are chained comparisons.
4. *Repeated Evaluation*: Loops: `while`...`end`, and `for`...`end`.
5. *Exception Handling*: `try`...`catch`, `error`, and `throw`. Exceptions are thrown when an unexpected condition has occurred.
6. `Tasks` (aka *Coroutines*), a generalized subroutine in Julia, which is a set of instructions stored in memory and invoked during execution. `yieldto` allow suspending and resuming of tasks.

The `for`...`end` loops iterate over iterables; iterable types include `Range`, `Array`, `Set`, `Dict`, and `AbstractString`. In loops you can use `in` instead of range generators
`while`...`end` loop lops while a condition *predicate* is `true`, and terminates when the predicate become `false`.

Sequential *iteration* is implemented by the `iterate` function [**?**]. Instead of mutating objects as they are iterated over, Julia iterators may keep track of the iteration state externally from the object. The return value from `iterate` is always either a tuple of a value and a state, or `nothing` if no elements remain.

Type system

Julia has a type system. Every *value* has a type; variables do not have types themselves. You can use the `typeof` function to get the type of a value: `abcd = "abcd"; typeof(abcd) # => String`.

Types are first-class values that can be used as arguments of functions and returned by functions. `DataType` is the type that represents types, including itself.

Julia's type system is *dynamic*. Still, it gains some of the advantages of static type systems by making it possible to indicate that certain values are of specific types and by type reasoning. This can greatly assist in generating efficient code, but even more significantly, it allows *method dispatch* on the types of function arguments to be deeply integrated with the language.

In Julia, a type is *concrete* if it can be instantiated, that is, some type `T` is concrete if there exists at least one value `v` such that `typeof(v) # => T`. *Abstract* types cannot be instantiated; they are used to create hierarchies of types, useful to generate fast code. Both abstract and concrete types can be parameterized by other types. They can also be parameterized by symbols and values of types s.t. `isbits` returns true.

The hierarchical relationships between types are explicitly declared. One particularly distinctive feature of Julia's type system is that *concrete types* may not *subtype each other*: all concrete types are *final*: may only have abstract types as supertypes.

Users can define new types. User-defined types are like *records* or *structs* in other languages. New types are defined using the `struct` keyword. Julia gives a default constructor as a function with the same name of the type. The default constructor's arguments are the *properties* of the type, in the order they are listed in the definition [?].

These struct-style types are *concrete types*. So, they can be instantiated, but cannot have *subtypes*. The other kind of types is `abstract` types defined as: `abstract` <*name*> `end`. Abstract types *cannot* be instantiated, but can have subtypes. `AbstractString`, as the name implies, is also an abstract type.

Type relations and hierarchy

The method `supertype(T::DataType)` returns the `DataType` father of `T`. The *subtype* operator is a predicate `T1 <: T2` that returns `true` when values of type `T1` are also of type `T2`. The *supertype* operator `T1 >: T2` is equivalent to `T2 <: T1`. Two examples follows (see also Figure **??**):

```
subtypes # => (generic function with two methods)
subtypes(Number) # => [Complex, Real]|
```

Types are used for documentation, optimizations, and dispatch. In Julia, they are not statically checked, but used by the JIT compiler to create faster code. The `::` operator may be used to attach type annotations to expressions and variables within the program. As a general rule, it is better not to use type annotation but let the compiler do it, so writing generic code makes also easier to interact with other packages.

The `abstract type` declares a type that *cannot* be instantiated and serves only as a node in the type graph, thereby describing sets of *related concrete types*: those concrete types which are their descendants. Abstract types form the conceptual hierarchy, which makes Julia's type system more than just a collection of object implementations. For example: `Number` has supertype `Any`, whereas `Real` is an abstract subtype of `Number` (see Figure **??**).

A `primitive` type declares a concrete type whose data consists only of a series of bits. Classic examples of primitive types are integers and floating-point values. Some examples of built-in primitive type declarations from Julia implementation:

```
primitive type Char 32 end
primitive type Bool <: Integer 8 end
```

Type declarations can be used in `global` scope, i.e., type annotations can be added to global variables to make accessing them type stable (about the importance of `type-stability` read [?]).

```
julia> x::Int = 10
10
julia> x = 3.5
ERROR: InexactError: Int64(3.5)
```

1.2 Functions and collections

The actual backbone of Julia programming are the data objects of *collection* types, including arrays, tuples, dictionaries, and sets. Objects of `Function` type have great significance in the language, since they denote any computable transformation from input data to output results of a computation.

1.2.1 Julia functions

In Julia, a function is an object that maps a tuple of argument values to a `return` value. All arguments to functions are passed by sharing (i.e., by memory address). Julia functions are not *pure* mathematical functions because they can alter and be affected by the global state of the program.

Definition 1.1 (Basic syntax for defining functions)

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

A *generic* function can be used for different types of its arguments. In simple words, whenever the function is called with arguments of a new type, the Julia compiler will generate a different version, called *method* for that function. The same happens when a function name is invoked with different number and types of arguments. See Julia Multiple Dispatch [**?**].

The keyword `function` creates new functions. Functions return the value of their last expression.

Definition 1.2 (Statement functions) There is a compact definition of functions, like in old Fortran. Here is the function definition with formal arguments, andt its application to actual argument values:

```
julia> f_add(x,y) = x + y
f_add (generic function with 1 method)

julia> f_add(2,3)
5
```

Definition 1.3 (Multiple return values) Functions can also return multiple values, as a tuple.

```
julia> f(x,y) = x+y, x-y, x*y
f (generic function with 1 method)

julia> a,b,c = f(2,4)
(6, -2, 8)
```

Definition 1.4 (Variable number of arguments) You may define functions whose $< head >$ takes a *variable* number of *positional* arguments:

```
function varargs(args...) < body > end
```

of course followed by a *body* of instructions and by the `end` token.

The token `...` is called a *splat*. We just used it to define a function *head*. The splat can also be used in a *function call*, where it will splat an array or tuple contents into the *argument* list:
`add(list...)` is equivalent to `add(5,6,7,8,9)` when `list = [5,6,7,8,9]`.

Definition 1.5 (Optional positional arguments) You may define functions with *optional* positional arguments. with an assigned *default* value, and hence not necessarily with an assigned value at runtime.

```
julia> function defaults(a, b, x=5, y=6)
           return "$a $b and $x $y"
       end
defaults (generic function with 3 methods)

julia> defaults(0, 0, y=10)
"0 0 and 5 10"
```

The `$` is used to interpolate values of variables or expressions into strings.

Definition 1.6 (keyword-optional arguments) You may also define functions that take keyword-optional arguments.

```
julia> function keyword_args(;k1=4, name2="hello")
            return Dict("k1" => k1, "name2" => name2)
       end
keyword_args (generic function with 1 method)
```

Note the `;` character before the optional arguments

Coding 1.2.1 *Function call* A corresponding function call follows, showing the parameter names `k1` and `name2` together with the assigned values, used to increase the code readability:

```
julia> keyword_args(; k1=4, name2="hello")
Dict{String, Any} with 2 entries:
  "name2" => "hello"
  "k1"    => 4
```

You can combine all kinds of arguments in the same function, with keyword ones defined in the last positions, following the `;` character. They can be invoked in any number and order, and possibly substituted by the default value. □

Functional programming style

Julia allows the programmer to use efficiently several traits of functional programming style, as it is shown in the following of this section.

Coding 1.2.2 In other words, Julia has *first class* functions. In the following scriptlet, the function `create_adder` returns an `adder` function:

```
function create_adder(x)
    adder = function(y)
        return x + y
    end
    return adder
end
```

It is also possible to name the internal function if desired:

```
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end
```

To be called, e.g., `add_10 = create_adder(10); add_10(3) #=> 13`.

Definition 1.7 (lambda syntax) The *lambda syntax* or ”stabby lambda syntax” is used to create *anonymous* functions: `(x -> x + 2)(3) # => 5`, where the lambda expression is `x -> x + 2`. The *arguments* are before the characters ” `->` “, and after the stab we have the *value-generating expression*.

```
function create_adder(x)
    y -> x + y
end
```

Of course, it is possible to use a tuple of arguments in the lambda form of functions: `((x,y) -> x + y)(2, 3) # => 5`. This function is identical to the `create_adder` implementation above.

Coding 1.2.3 Even more, like a proper functional language, with curried parameters, we may have:

```
add = (x -> y -> x + y);
add(2)(3) # => 5.
```

A curried function is a function which takes multiple parameters one at a time, by taking the first argument, and returning a series of functions which each take the next argument until all the parameters have been fixed, and the function application can complete, at which point, the resulting value is returned. Note that the number of arrows equals the number of applications.□

Julia higher-order functions

The `map()`, `filter()`, and `reduce()` functions are three fundamental higher-order functions that are found in almost every programming language today. In Julia we have this syntax for built-in higher-order functions:

```
map(add_10, [1,2,3])                # => [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) # => [6, 7]
reduce(*, [2; 3; 4]; init=-1)       # => -24
```

`Reduce` takes two arguments — a function `f` and a collection `A`. The function `f` must take two arguments, and then `reduce` goes through the collection and one-element-at-a-time it updates `result = f(result, elt)`. The keyword argument `init` is the initial value to use in the reductions. For `+`, `*`, `max`, and `min` the `init` argument is optional.

```
julia> a = reshape(Vector(1:16), (4,4))
  4×4 Matrix{Int64}:
   1  5   9  13
   2  6  10  14
   3  7  11  15
   4  8  12  16
julia> reduce(max, a, dims=2)
  4×1 Matrix{Int64}:
   13
   14
   15
   16
julia> reduce(max, a, dims=1)
  1×4 Matrix{Int64}:
   4  8  12  16
```

You may also consider using `foldl` or `foldr`, with fixed associativity direction: `foldr(op, itr; [init])` is like `reduce`, but with guaranteed *right* associativity. Or `foldl(op, itr; [init])` is like `reduce`, but with guaranteed *left* associativity, `foldr(*, 1:5; init=1) # => 120` in this case also `foldr(*, 1:5; init=1) # => 120`. Such recursive functionals transform any binary operator into an *n*-ary operator and might implement many geometric programming patterns.

Remark 1.1 When redefining a method or adding new methods, realizing that these changes don't take effect *immediately* is essential.

This is key to Julia's ability to statically infer and compile code to run fast without the usual JIT tricks and overhead. Indeed, any new method definition won't be immediately visible to the current runtime environment, including `Tasks` and `Threads` (and any previously defined `@generated` functions, an excellent *metaprogramming* tool [**?**]).

1.2.2 Collections

Julia `Base` package contains a variety of functions and macros suitable for performing scientific and numerical computing but is as broad as many general-purpose programming languages. Additional functionality is available from a growing collection of available packages.

The proper Julia collections are arrays, tuples, dictionaries, and sets. All (but sets) can be accessed by integer indices. User-defined collections must satisfy the `Iterable Collections` protocol.

Beware, Julia indexes everything from `1` (like MATLAB and Fortran), not `0` (like most languages, including C, C++, Rust, and Java). Or else, iterating over strings is recommended (`map`, `for` loops, etc). The notation $`(expr)` can be used for interpolation of a value inside a string, making complex printing very easy. Note that parenthesis `()` are *mandatory* when `expr` is not a single token.

Arrays

Julia provides a complete collection of basic arithmetic and bitwise operators across all of its numeric primitive types and portable, efficient implementations of a comprehensive collection of standard mathematical functions.

`AbstractArray{T, N}` is the supertype for N-dimensional arrays (or array-like types) with elements of type `T`; array and other types are subtypes of this.

`AbstractVector{T}` is the supertype for one-dimensional arrays (or array-like types) with elements of type `T`. It is an alias for `AbstractArray{T,1}`.

`AbstractMatrix{T}` is the supertype for two-dimensional arrays (or array-like types) with elements of type `T`. It is an alias for `AbstractArray{T,2}`.

A broadcast dot operator (`.`) applies any Julia operator or function to all the elements of a collection: `f.[a,b,c] # => [f(a),f(b),f(c)]`.

Tuples

In Julia, `typeof(Tuple) # => DataType` is an *immutable* collection of distinct values of the same or different datatypes separated by commas.

Tuples are a *heterogeneous* collection of values. Tuples are more like arrays in Julia, except that arrays only take values from the same datatype. The values in a tuple *cannot be changed* because tuples are immutable. The whole tuple value in a variable can only be *replaced* by a new tuple value.

The sequence of values stored in a tuple can be of any type, and integers index them. Although not needed, defining a tuple using *parentheses* around the sequence of values is expected. This helps in understanding the Julia tuples more easily.

Coding 1.2.4 The `tuple` function returns a tuple from given objects:

```
tuple(1, 'b', pi) # => (1, 'b', π).
```

Coding 1.2.5 The function `ntuple(f::Function, n::Integer)` creates a tuple of length n, computing each element as `f(i)`, where `i` is the index of the element:

```
ntuple(i -> 2i, 4) # => (2, 4, 6, 8).
```

Dictionary

A simple look-up table is a helpful way of organizing many data types: given a single piece of information, such as a number, string, or symbol, called the *key*, what is the corresponding data *value*? For this purpose, Julia provides the `Dictionary` object, called `Dict` for short. It is an ”associative collection” because it associates keys with values [**?**].

Definition 1.8 (Dict is the standard dictionary type) Its implementation uses `hash` as the hashing function for the key and `isequal` to determine equality. Redefine these two functions for *custom types* to override how they are stored in a hash table. Any `hash` function must compute an integer hash code such that `isequal(x,y)` implies `hash(x) == hash(y)`.

Dictionaries can be created by passing *pair objects* constructed with `=>` to a `Dict` *constructor*: `Dict("A"=>1, "B"=>2)`. This call will attempt to infer type information from the keys and values (i.e., this example creates a `Dict{String, Int64}`. To explicitly specify types use the syntax `Dict{KeyType,ValueType}(...)`.

Coding 1.2.6 Construction syntax directly using pairs `key => value`:

```
julia> Dict{String, Int32}(“A”=>1, "B"=>2)   #=
Dict{String, Int32} with 2 entries:
  "B" => 2
  "A" => 1          =#
```

Coding 1.2.7 Dictionaries may also be created using *generators*.

```
julia> f = i->2i; Dict(i => f(i) for i=1:10)   #=
Dict{Int64, Int64} with 10 entries:
  5 => 10
  4 => 8
  ⋮ => ⋮             =#
```

Remark 1.2 Let us notice that the pairs in a `Dict` are not ordered according to the generation sequence. `OrderedDict` in `DataStructures` package is used for this purpose.

Given a dictionary `D`, the syntax `D[x]` returns (in reading) the value associated to key `x` (if it exists) or throws an error, and `D[x] = y` stores (in writing) the key-value pair `x => y` in `D` (replacing any existing value for the key `x`).

Multiple arguments to `D[...]` are converted to `tuples`; for example, the syntax `D[x,y]` is equivalent to `D[(x,y)]`, i.e., it refers to the value keyed by the tuple `(x,y)`.

Set

Definition 1.9 Sets are *mutable containers* that provide fast membership testing. `Set{T} <: AbstractSet{T}`.

`Set` datatype enjoys efficient implementations of set operations such as `in`, `union`, and `intersect`. Elements in a `Set` are *unique*, as determined by the elements' definition of `isequal`. The order of elements in a `Set` is an *implementation detail* and cannot be relied on.

Coding 1.2.8 Some `Set` examples follow:

```
julia> s = Set("aaBca")     #=
Set{Char} with 3 elements:
  'a'
  'c'
  'B'       =#
julia> push!(s, 'b')       #=
Set{Char} with 4 elements:
  'a'
  'c'
  'b'
  'B'       =#
julia> union([4 2 3 4 4], 1:3, 3.0)     #=
4-element Vector{Float64}:
  4.0
  2.0
  3.0
  1.0       =#
```

Let note the type *promotion* of all `Set` elements to the same type `Float64`.□

1.3 Matrix computations

A *primitive type* is a concrete type whose data consists of plain old bits. Classic examples of primitive types are integers and floating-point values. Julia's primitive *numeric* types called *bits types* (integers, both signed and unsigned, Booleans, and floats) are `Int8`, `UInt8`, `Int16`, `UInt16`, `Int32`, `UInt32`, `Int64`, `UInt64`, `Int128`, `UInt128`, `Bool`, `Float16`, `Float32`, `Float64`, depending on the number of bits. Additionally, full support for `Complex` and `Rational` numbers is built on top of these primitive numeric types.

Vector and Matrix Types

In Julia, the `Array` type holds both "bits" values as well as "boxed" values. Boxed variables are *heap-allocated* and tracked by the garbage collector.

The distinction is whether the value itself is stored *inline* (in the directly allocated memory of the array, i.e., without following pointers to it) or if the array's memory is simply a collection of pointers to objects allocated elsewhere. Regarding performance, accessing values inline is a substantial advantage over following a *pointer* to the actual values.

Julia provides a first-class array implementation without treating arrays in any special way. The array library is implemented almost entirely in Julia itself and derives its performance from the compiler, just like any other code written in Julia.

The *type* `Vector{T} <: AbstractVector{T}` is a 1-dimensional *dense array* with elements of type `T`, often used to represent a mathematical vector. It is an alias for `Array{T,1}`.

Of course, the *method* `Vector{T}(undef,n)` constructs an *uninitialized* object `Array{T,1}` of length `n`. Analogously, we have the `Base.Matrix` as the alias `Matrix` in `Base` package:

1. the parameterized *type* `Matrix{T} <: AbstractMatrix{T}`, is a two-dimensional dense array with elements of type `T`, often used to represent a mathematical matrix, where stands for `Array{T,2}`;
2. the *method* `Matrix{T}(undef,m,n)` which construct an *uninitialized* `Array{T,2}` of size $m \times n$.

Vectors in Julia

We see how to create and manipulate vectors in Julia and how Julia notation differs from mathematical notation.

To create the 3-vector $x = (8, -4, 3.5) = \begin{pmatrix} 8 \\ -4 \\ 3.5 \end{pmatrix}$, use: `x = [8,-4,3.5]`, but `x = [8;-4;3.5]` also works. Be careful for similar-looking expressions, because `(8,-4,3.5)`, `[8,-4,3.5]` and `[8 -4 3.5]` are not equivalent in Julia. They are a tuple, a column matrix, and a row matrix, respectively.

To get an integer range from i to j (for $i \leq j$), let's use a colon iterator `i:j`. The assignment `x = collect(1:10)` returns the array `x`. To specify an increment size, add an argument.

```
x = collect(1:0.1:10)' #= >
[1.0 1.1 1.2 1.3 1.4 … 9.4 9.5 9.6 9.7 9.8 9.9 10.0].   =#
```

The range from `1` to `10` with `0.1` step size is given above. Note the apex for transposition, which is needed for typographical reasons here.

1. indexes run from `1` to `n = length(x)`, and x_2 is `x[2]`
2. you can also *set an element*, e.g., `x[3] = 10.5`;
3. use a range to select more than one element; `x[2:3]` selects the second and third elements ;
4. `x[end]` selects the last element;
5. to select the even significant elements of `x` use

```
x[1:2:end]' # => [1.0 1.2 1.4 … 9.6 9.8 10.0].
```

To form a `stacked Vector` made by vectors `a=[1;2]` and `b=[3;4;5]` use `[a...,b...]` or `[a;b]`.

The expression `[a,b]` would return a `Vector{Vector{Int64}}` value with 2 elements of type `Vector{Int64}`. Stacked vectors can be used as list of lists. To access an element in `w = [a,b]` use `w[2][2] # => 4`

Many more vector operations are defined in `Base` package. In Julia, a scalar and a vector can be added using the dot (broadcast) operator. The scalar is added to each entry of the vector: `[2,4,8] .+ 3 # => [5,7,11]`.

Scalar-vector multiplication uses `*` because the operator is linear: `[2,4,8] * 3 # => [6,12,24]`. Both expressions are commutative.

In Julia syntax, like in MATLAB and differently from Python: `+` and `*` operate on congruent arrays, otherwise promote the arguments to same type.

Matrices in Julia

Julia provides a very simple notation to create matrices. A matrix can be created using the following notation: `A = [1 2 3; 4 5 6]`. Spaces separate entries in a row and semicolons separate rows. We can also get the size of a $m \times n$ matrix using `size(A) = (m,n)`. Block matrices are easily generated by the same notation, as long as matrix or vector blocks of consistent size are used.

In addition to (and as part of) its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations which can be loaded with using `LinearAlgebra`. Basic operations, such as `tr`, `det`, and `inv` are all supported in `LinearAlgebra` package. As well as other useful operations [?], such as finding `eigenvalues` or `eigenvectors`:

```
A = [-4. -17.; 2. 2.]   #=
2×2 Matrix{Float64}:
 -4.0  -17.0
  2.0    2.0    =#
eigvals(A)  #=
2-element Vector{ComplexF64}:
 -1.0 - 5.0im
 -1.0 + 5.0im   =#
eigvecs(A)  #=
2×2 Matrix{ComplexF64}:
  0.945905-0.0im        0.945905+0.0im
 -0.166924+0.278207im  -0.166924-0.278207im =#
```

1.4 Linear algebra and sparse arrays

In addition to its support for multi-dimensional arrays, Julia provides native implementations of many common and useful linear algebra operations, which can be loaded by `using` the package. A simple example is given. Note the promotion of values in matrix `A`:

```
using LinearAlgebra
A = [3. 9 8; 5 9 1; 4 8 2]  #=
3x3 Matrix{Int64}:
 3.0  9.0  8.0
 5.0  9.0  1.0
 4.0  8.0  2.0  =#
A'      #=
3x3 adjoint(::Matrix{Int64}) with eltype Int64:
 3.0  5.0  4.0
 9.0  9.0  8.0
 8.0  1.0  2.0  =#
```

Basic operations, such as `tr`, `det`, `rank`, and `inv` are supported by the package `LinearAlgebra`:

```
tr(A)   # => 14.0
det(A)  # => 7.999999999999996
rank(A) # => 3
inv(A)          #=
3x3 Matrix{Float64}:
  1.25   5.75  -7.875
 -0.75  -3.25   4.625
  0.5    1.5   -2.25    =#
```

The inner product $a^{\top}b$ is written as `LinearAlgebra.dot(a,b)`. `Vector`s `a` and `b` must have the same `length`.

The norm $x = \sqrt{}(x_1^2 + x_2^2 + \cdots + x_n^2)$ is written `LinearAlgebra.norm(x)`.

The distance of two vectors $dist(x,y)$ = y - x is `LinearAlgebra.norm(y-x` `)`. Of course, the qualification `LinearAlgebra.` is not explicitly needed, after declaration, for exported package symbols. The function Root Mean Square: $rms(x) = \sqrt{}((x_1^2 + \cdots + x_n^2)/n) = x/\sqrt{n}$ can be expressed as:

```
rms(x) = norm(x)/sqrt(length(x))    #=  function definition
rms (generic function with 1 method)    =#
x    #=
1x2 Array{Float64,2}:
 0.543101  0.335506     =#
rms(x)                              #=  function application
0.45139931240367087     =#
```

From Rosetta Code [?] we report several ways of implementing/using the $rms(x)$ function. In fact, there are a variety of ways to do this via built-in functions in Julia, given an array `A = rand(10)` of values.

The formula can be implemented directly using the comma broadcast:

```
sqrt(sum(A.^2)/length(A))
```

or shorter by `using Statistics` package: `sqrt(mean(A.^2))`. The implicit allocation of a new array by `(A.^2)` can be avoided by using `sum` as a higher-order function: `sqrt(sum(x -> x*x, A)/length(A))`. Of course, one can also use an explicit loop for near–C performance:

```
function rms(A)
   s = 0.0
   for a in A
      s += a*a
   end
   return sqrt(s/length(A))
end
```

Potentially even better is to use the built-in `norm` function, which computes the square root of the sum of the squares of the entries of `A` in a way that avoids the possibility of spurious floating-point overflow (if the entries of `A` are so large that they may overflow if squared): `norm(A)/sqrt(length(A))`.

To solve a linear system, $Ax = b$ by `x = LinearAlgebra.inv(A) * b` is not recommended for big `A` matrices. A better approach would be to compute `x = A \ b`, which is equivalent to

```
qr(A, Val(true)) \ b
```

This expression uses a pivoted QR factorization to solve the least-squares problem, which is much more accurate than the normal-equations approach for poorly conditioned `A`.

In addition, Julia provides many factorizations which can be used to speed up problems such as linear solve or matrix exponentiation by pre-factorizing a matrix into a form more amenable (for performance or memory reasons) to the problem. In `factorize(A)` documentation [?] we find:

Compute a suitable factorization of `A`, based upon the type of the input matrix. `factorize` checks `A` to see if it is `symmetric/triangular/`etc. if `A` is passed as a generic matrix. `factorize` checks every element of `A` to verify/rule out each property. It will short-circuit as soon as it can rule out symmetry/triangular structure. The return value can be reused for efficient solving of multiple systems. For example:

```
A = factorize(A);  x = A \ b.
```

Furthermore,

```
Y = A \ C.
```

Technically, two matrices cannot be divided, but if you remember the basic concept of division of two fractions, then second fraction remains the same but the first fraction gets inverted and multiplied to the former fraction for performing the division. Example with rational numbers:

```
(4//3 \ 5//2 == 3//4 * 5//2 == 15//8) == true.
```

Sparse Arrays

In numerical analysis and scientific computing, a sparse matrix is a matrix with many zero elements. In other words, the sparse array is an array where many elements have a recurring value, typically zero. It is, therefore, possible to use a better storage model, storing only the non-zero values. There are several storage schemes for numeric sparse vectors and sparse matrices.

The most straightforward scheme is called *storage by triples* `(i,j,v)`, where some suitable data structure supplies the row index, the column index, and the corresponding value of every non-zero. In Julia, this is implemented in the `SparseArray` package as input method `sparse()` with input parameters `(I, J, V)`: two arrays of integer indices and one array of bits type for values.

Typical implementation schemes for sparse matrices are the CSR (compressed sparse row) and CSC (compressed sparse column). Julia uses, by now, only the CSC scheme. Julia supports sparse vectors and sparse matrices in the `SparseArrays stdlib` module. Sparse arrays contain enough zeros that storing them in a special data structure leads to savings in space and execution time compared to dense arrays.

As said above, n Julia, sparse matrices are stored in the *Compressed Sparse Column* (CSC) format. Julia sparse matrices have the type `SparseMatrixCSC {Tv, Ti}`, where `Tv` is the type of the stored values, and `Ti` is the integer type for storing column pointers and row indices.

The internal representation of type `SparseMatrixCSC` is as follows:

```
struct SparseMatrixCSC{Tv,Ti<:Integer} <:
    AbstractSparseMatrixCSC{Tv,Ti}
    m::Int                  # Number of rows
    n::Int                  # Number of columns
    colptr::Vector{Ti}      # Column j in colptr[j]:(colptr[j+1]-1)
    rowval::Vector{Ti}      # Row indices of stored values
    nzval::Vector{Tv}       # Stored values, typically nonzeros
end
```

In some applications, storing explicit zeros in a `SparseMatrixCSC` is convenient. The `nnz` function returns the number of elements explicitly stored in the sparse data structure, including non-structural zeros. To count the exact number of numerical nonzeros, use `zcount(!iszero, x)`, which inspects every stored element of a sparse matrix. `dropzeros`, and the in-place `dropzeros!`, can be used to remove stored zeros from the sparse matrix.

Sparse vectors are stored in a close analog to a compressed sparse column format for sparse matrices. In Julia, sparse vectors have the type `SparseVector{Tv, Ti` where `Tv` is the type of the stored values and `Ti` is the integer type for the indices. The internal representation is as follows:

```
struct SparseVector{Tv,Ti<:Integer} <: AbstractSparseVector{Tv,
    Ti}
    n::Int                  # Length of the sparse vector
    nzind::Vector{Ti}    # Indices of stored values
    nzval::Vector{Tv}    # Stored values, typically nonzeros
end
```

The `sparse` function is often handy for constructing sparse arrays. For example, to construct a sparse matrix, we can input a vector `I` of row indices, a vector `J` of column indices, and a vector `V` of stored values (this is also known as the `COO` (coordinates) format). `sparse(I,J,V)` then returns a sparse matrix `S` such that `S[I[k], J[k]] = V[k]`. The equivalent sparse vector constructor is `sparsevec`, which takes the (row) index vector `I` and the vector `V` with the stored values and constructs a sparse vector `R` such that `R[I[k]] = V[k]`.

The inverse of the `sparse and sparsevec` functions is `findnz`, which retrieves the inputs used to create the sparse array. `findall(!iszero, x)` returns the cartesian indices of non-zero entries in `x` (including stored entries equal to zero).

Details and examples can be found in the Sparse Vectors and Matrices section of the standard library reference [**?**].

1.5 Parallel and distributed computing

Julia makes easier the implementation of complex algorithms by providing native support for concurrency and fast computation by design. Implementing concurrency and multithreading of computations in Julia is mainly a matter of following some style rules and inserting a few macros in the program code.

Parallelism at all levels, from instruction execution to distributed computation, is mostly implicit. Task-based control flows for parallel execution are natively provided, like booth cooperative multitasking and thread-based preemptive multitasking.

Of course, Julia's multithreading model offers the ability to schedule tasks simultaneously on more than one thread or CPU core, sharing memory.

1.5.1 Parallel Programming

There are three steps to program parallelization. First, you identify a decomposition into tasks to be computed concurrently.

If the number of tasks is too large compared to the number of threads, the overhead of scheduling them could slow down the computation. No free lunches. The third step is to figure out how to map tasks to threads [**?**].

According to Julia manual, the language supports four major classes of concurrent and parallel programming [**?**], also known as models of parallel computing: (a) asynchronous ”tasks”, or coroutines; (b) multi-threading; (c) distributed computing; (d) GPU computing.

Asynchronous tasks (coroutines)

Coroutines are program components that allow execution to be suspended and resumed, generalizing subroutines (computation with effects as opposed to functions returning a value) for *cooperative* multitasking. This one is a style of multitasking in which the operating system never executes a context switch at system level from a running process to another process. Conversely, tasks voluntarily yield control periodically or when idle or logically blocked.

Tasks are a control flow feature that allows computations to be suspended and resumed in a flexible manner. Julia tasks allow suspending and resuming computations for I/O, event handling, producer-consumer processes, and similar patterns, in order to run multiple applications concurrently.

Objects of type `Task` can be created by the macro `@task x`, where `x` is any expression, usually `begin; ...; end`. The `Task` object can be assigned to a variable: `t = @task x`. After creation, the task must be started; it is started by calling `schedule(t)`, where it is added to a queue of tasks waiting resources for execution. It is common to create a task and `schedule` it immediately. The macro `@async` is provided for that purpose: `@async x` is equivalent to `schedule(@task x)`.

The macro `@async` wrap an expression in a `Task` and add it to the local machine's scheduler queue. Values can be interpolated into `@async` via `$`, which copies the *value* directly into the constructed underlying *closure*. This allows you to insert the value of a variable, *isolating* the asynchronous code from *changes* to the variable's value in the current task.

It is strongly encouraged to favor `Threads.@spawn` over `@async` always. This is because a use of `@async` disables the migration of the parent task across worker threads in the current implementation of Julia. Thus, seemingly innocent use of `@async` in a library function can have a large impact on the performance of very different parts of user applications [**?**].

The macro `@sync` waits until all lexically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@distributed` are complete. In practice it works like a parallel barrier in other languages. One of the simplest examples is using `@sync/@async` for non-blocking I/O. For example, you want to download 10 web pages. If you do it in a simple blocking loop, most of the time Julia does nothing but waits for network:

```julia
urls = ["https://discourse.julialang.org/" for i=1:10]
results = Vector(10)
@time for (i, url) in enumerate(urls)
    results[i] = Requests.get(url)
end
```

But network I/O in Julia is non-blocking, so you can use task machinery to accelerate:

```
@sync for (i, url) in enumerate(urls)
    @async results[i] = Requests.get(url)
end
```

In the following example each `println` request is run in a separate task using `Threads.@spawn` and then wait for the result of all of them because of `@sync`. When one of these tasks encounters I/O operation, it gives away control so that other tasks could use CPU. When I/O operations are finished, the task is resumed via `@sync`.

```
julia> Threads.nthreads() # => 4
```

```
julia> @sync begin
    Threads.@spawn
      println("Thread-id $(Threads.threadid()), task 1")
    Threads.@spawn
      println("Thread-id $(Threads.threadid()), task 2")
end;
# =>
Thread-id 3, task 1
Thread-id 1, task 2
```

Tasks communicate via `Channels`. While strictly not parallel computing, Julia lets you schedule `Task`s on several threads. When a piece of computing work (in practice, executing a particular function) is designated as a `Task`, it becomes possible to interrupt it by *switching* to another Task. The original `Task` can later be resumed, at which point it will pick up right where it left off.

At first, this may seem similar to a function call. However there are two key differences. First, switching tasks does not use any space, so any number of task switches can occur without consuming the call stack. Second, switching among tasks can occur in any order, unlike function calls, where the called function must finish executing before control returns to the calling function [?].

Julia tasks are not threads. They are coroutines which can be scheduled asynchronously on a single thread or multiplexed onto a thread pool. All I/O in Julia is non-blocking and yields to the scheduler, so it is more of a cooperative situation.

Data parallelism

For data parallelism, a higher-level description is appropriate. It also helps you write more reusable code; e.g., using the same code for single-threaded, multi-threaded, and distributed computing

In particular, it is important to use libraries that help you describe *what* to compute rather than *how* to compute. Practically, it means to use generalized form of `map` and `reduce` operations and learn how to express your computation in terms of them. Luckily, if you already know how to write *iterator comprehensions*, there is not much more to learn for accessing a large class of data parallel computations. [?].

Like how multi-threading is setup, you need to setup multiple worker processes to get speedup. You can start julia with `-p auto` and with `-t auto` to maximize the number of processes and tasks *sharing memory*.

`Mapping` is probably the most frequently used function in data parallelism. In sequential code we have: `map(f, vect)` which results in `[f(vect[i] for i in vect]`; but Julia's standard library `Distributed.jl` contains a function `pmap` as a distributed version of `map`, so that you can write using `Distributed; pmap(f,vect)`. Also you can use `Folds` library and write: `using Folds; Folds.map(f,vect)` with good speedups on large collections.

```
pmap(f, [::AbstractWorkerPool], c...; distributed=true, [...])
```

Transform collection `c` by applying `f` to each element using available workers and tasks. For multiple collection arguments, apply `f` elementwise. Note that `f` must be made available to all worker processes; see Code Availability and Loading Packages for details. If a worker pool is not specified, all available workers, i.e., the default worker pool is used. By default, `pmap` distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`.

Julia's *iterator comprehension* syntax is a powerful tool for composing mapping, filtering, and flattening. Recall that sequential mapping can be written as an array or iterator comprehension:

```
b1 = map(x -> x + 1, 1:3)          # => [2,3,4]
b2 = [x + 1 for x in 1:3]          # array comprehension
b3 = collect(x + 1 for x in 1:3)   # iterator comprehension
@assert b1 == b2 == b3             # => true
```

The *iterator comprehension* can be executed with threads via `Folds.collect` [?]:

```
b4 = Folds.collect(x + 1 for x in 1:3)
@assert b1 == b4                     # => true
```

Functions such as `sum`, `prod`, `maximum`, and `all` are the examples of *reduction* (aka `fold`) that can be parallelized. Using `Folds.jl`, a sum of an iterator created by the comprehension syntax can easily be parallelized by

```
d = Folds.sum(x + 1 for x in 1:3).
```

Multi-threading

Multithreading enables programmers to speed up their programs by taking advantage of concurrent execution of multiple threads, with each thread assigned to a different CPU core. On the surface, this type of programming may seem easy, but in practice it can be difficult to ensure correctness and to obtain signficant speedup.

Julia supports two different models for multithreaded programming: loop parallelism with the `@threads` macro and task parallelism with the
`Threads.@spawn` macro, which is low-level basic construct. Julia's multithreading provides the ability to schedule Tasks simultaneously on more than one thread or CPU core, *sharing memory*. This is usually the easiest way to get parallelism on one's PC or on a single large multi-core server. Julia's multi-threading is composable. When one multi-threaded function calls another multi-threaded function, Julia will schedule all the threads globally on available resources.

Although Julia's threads can communicate through shared memory, it is notoriously difficult to write correct and data-race free multi-threaded code. Julia's Channels are thread-safe and may be used to communicate safely. The best way to ensure data-race freedom is to acquire a lock around any access to data that can be observed from multiple threads.

By default, Julia starts up with a single thread of execution.
`Threads.nthreads()` returns the number of thrads, setted by parameter -t when starting Julia, for example `-t auto` or `--threads n`. The function `Threads.threadid()` returns the integer `id` of the current thread.

Julia supports parallel loops using the Threads.@threads macro. This macro is affixed in front of a for loop to indicate to Julia that the loop is a multi-threaded region. Julia supports accessing and modifying values atomically, that is, in a thread-safe way to avoid race conditions.

When a program's threads are busy with many tasks to run, tasks may experience delays which may negatively affect the responsiveness and interactivity of the program. To address this, you can specify that a task is `interactive`.

External libraries, such as those called via `ccall`, pose a problem for Julia's task-based I/O mechanism. If a C library performs a blocking operation, that prevents the Julia scheduler from executing any other tasks until the call returns.

There are a few specific limitations and warnings to be aware of when using threads in Julia [?].

1.5.2 Multiprocessing and Distributed Computing

Most modern computers possess more than one CPU, and several computers can be combined together in a cluster. Harnessing the power of these multiple CPUs allows many computations to be completed more quickly.

There are two major factors that influence performance: the speed of the CPUs themselves, and the speed of their access to memory. In a cluster, it's fairly obvious that a given CPU will have fastest access to the RAM within the same computer (node). Perhaps more surprisingly, similar issues are relevant on a typical multicore laptop, due to differences in the speed of main memory and the cache [?].

Distributed computing runs multiple Julia processes with separate memory spaces. These can be on the same computer or multiple computers. The `Distributed` standard library provides the capability for remote execution of a Julia function. With this basic building block, it is possible to build many different kinds of distributed computing abstractions. Packages like `DistributedArrays.j` are an example of such an abstraction. On the other hand, packages like `MPI.jl` and `Elemental.jl` provide access to the existing `MPI` ecosystem of libraries.

An implementation of distributed memory parallel computing is provided by module `Distributed` as part of the standard library shipped with Julia.

Consequently, a good multiprocessing environment should allow control over the "ownership" of a chunk of memory by a particular CPU. Julia provides a multiprocessing environment based on message passing to allow programs to run on multiple processes in separate memory domains at once [?].

Distributed programming in Julia is built on two primitives: *remote references* and *remote calls*. A remote reference is an object that can be used from any process to refer to an object stored on a particular process. Remote references come in two flavors: `Future` and `RemoteChannel`. A remote call is a request by one process to call a certain function on certain arguments on another (possibly the same) process.

Much easier is parallel programming in Julia by making use of
`DistributedArrays.jl`. This great computational abstraction uses the stdlib `Distributed` to implement a `Global Array` interface. In fact, large computations are often organized around large arrays of data. In these cases, a particularly natural way to obtain parallelism is to distribute arrays among several processes.

This combines the memory resources of multiple machines, allowing use of arrays too large to fit on one machine. Each process can read and write to the part of the array it owns and has read-only access to the parts it doesn't own. This provides a ready answer to the question of how a program should be divided among machines.

A `DArray` is distributed across a *set of workers*. Each worker can read and write from its local portion of the array and each worker has read-only access to the portions of the array held by other workers. Julia distributed arrays are implemented by the `DArray` type. A `DArray` has an element type and dimensions just like an `Array`. A DArray can also use arbitrary array-like types to represent the local chunks that store actual data. The data in a `DArray` is distributed by dividing the index space into some *number of blocks* in each dimension.

By using `DistributedArrays`, common kinds of arrays can be constructed with distributed data structures. E.g., `d = DistributedArrays; d.zeros`, `d.ones`, `d.rand`, `d.randn`, `d.fill`. The constructor that buils a distributed array is:

```
DArray(init, dims, [procs, dist])
```

1. the parameter `init` is a *function* that accepts a tuple of index ranges. This function should allocate a local chunk of the distributed array and initialize it for the specified indices.
2. `dims` is the overall size of the distributed array.
3. `procs` optionally specifies a vector of process `ID`s to use. If unspecified, the array is distributed over *all* worker processes only. Typically, when running in distributed mode, i.e., `nprocs() > 1`, this would mean that no chunk of the distributed array exists on the process hosting the interactive julia prompt.
4. `dist` is an integer vector specifying how many chunks the distributed array should be divided into in each dimension.

For example, the `dfill` function that creates a distributed array and fills it with a value `v` is implemented as:

```
dfill(v, args...) = DArray(I->fill(v, map(length,I)), args...)
```

1.5.3 Programming the GPU

While *kernel* functions for GPU are usually written in a C/C++ dialect, the Julia GPU compiler provides the ability to run Julia code *natively* on GPUs. There is also a rich ecosystem of Julia packages that target GPUs. The `JuliaGPU.org` website provides a list of capabilities, supported GPUs, related packages and documentation. Julia has several packages for programming GPUs, each of which support various programming models.

`JuliaGPU` is a Github organization created to unify the many packages for programming GPUs in Julia, in order to perform high-performance GPU programming in a high-level language.

JuliaGPU is vendor-neutral, and some content is available for all supported GPU back-ends. Of course JuliaGPU embraces also vendor-specific tools and APIs. At 2021 JuliaCon workshop, they demonstrated the use of three major GPU programming packages: `CUDA.jl` for Nvidia GPUs, `AMDGPU.jl` for AMD GPUs, and `oneAPI.jl` for Intel GPUs. The various approaches for programming GPUs with these packages, range from generic array operations that focus on ease-of-use, to hardware-specific kernels for when performance matters.

GPU functions (*kernels*) are inherently parallel, so writing GPU kernels is at least as difficult as writing low-level parallel CPU code, but the difference in hardware adds quite a bit of complexity. Most algorithms will need arrays to manage all their data, which calls for a good GPU array library.

GPUArrays.jl

`GPUArrays.jl` is the counterpart of Julia's `AbstractArray` interface, but for GPU types. It is not intended for the end-user, which should only use one of the packages that builds on GPUArrays.jl, such as `CUDA.jl` (Nvidia), `oneAPI.jl` (Intel), `AMDGPU.jl` (AMD), or `Metal.jl` (Apple), for different hardware support.

`GPUArrays` is an *abstract interface* for GPU computations. Think of it as the `AbstractArray` interface in Julia `Base` but for GPUs. It allows you to write generic julia code for all GPU platforms and implements common algorithms for the GPU. Like Julia `Base`, this includes `BLAS` wrapper, `FFT`s, `maps`, `broadcasts` and m`apreduces`. So when you inherit from `GPUArrays` and overload the interface correctly, you will get a lot of functionality for free.

It is important to remark that Julia allows you to write both GPU *kernels* and *surrounding code* in Julia itself, while running on most GPU hardware.

In complex analysis, the *Julia set* [**?**] of a holomorphic function consists of all those points whose behavior after repeated iterations of the function is chaotic, in the sense that it can change drastically following a small initial perturbation. As one can see in [**?**], the computational example of *Julia set* with Julia (!) strongly motivates why one should move big array computations to the GPU. For large arrays one gets a solid 60-80x speed-up by moving the calculation to the GPU. Getting this speed-up was as simple as converting the Julia `array` to a `GPUArray`.

CUDA and OpenCL

There is a model gap between `CUDA` and `OpenCL` parallel programming, which are the dominant frameworks used to write low-level GPU code. OpenCL is a heterogeneous programming platform that allows applications to run across multiple platforms, including CPUs, GPUs, and other specialized hardware. Conversely, CUDA is a software framework specifically designed to run computations on Nvidia's GPUs.

While `CUDA.jl` only supports Nvidia hardware, `OpenCL.jl` supports all hardware but "is a bit rough around the edges". One needs to decide what to use, and will get pretty much stuck with that decision [**?**].

One might think that the GPU performance suffers from being written in a dynamic language like Julia, but Julia's GPU performance should be pretty much on par with the raw performance of CUDA or OpenCL. Tim Besard, the creator of Julia GPU compiler, did a great job at integrating the LLVM Nvidia compilation pipeline to achieve the same or sometimes even better performance as CUDA C code [**?**].

CUDA programming in Julia

The `CUDA.jl` package is the main entry point for programming NVIDIA GPUs in Julia. The package makes it possible to do so at various abstraction levels, from easy-to-use arrays down to hand-written kernels using low-level CUDA APIs. The following is synthesized from Reference [**?**] that you are invited to read.

Julia has first-class support for GPU programming: you can use high-level abstractions or obtain fine-grained control, all without ever leaving your favorite programming language. The purpose of this tutorial is to help Julia users take their first step into GPU computing. In this tutorial, you'll compare CPU and GPU implementations of a simple calculation, and learn about a few of the factors that influence the performance you obtain.

We can perform GPU computations at a high level using the `CuArray` type, without explicitly writing a kernel function:

```julia
using CUDA
x_d = CUDA.fill(1.0f0, N) # vector stored on GPU filled of 1.0 (
    Float32)
y_d = CUDA.fill(2.0f0, N) # vector stored on GPU filled of 2.0
```

Here the `d` means `device`, in contrast with `host`. Now let's do the increment:

```julia
y_d .+= x_d
@test all(Array(y_d) .== 3.0f0) # => Test Passed
```

The statement `Array(y_d)` moves the data in `y_d` back to the host for testing. If we want to benchmark this, let's put it in a function:

```
function add_broadcast!(y, x)
    CUDA.@sync y .+= x
    return
end

add_broadcast! # => (generic function with 1 method)
@btime add_broadcast!($y_d, $x_d) # => 67.047 μs
  (84 allocations: 2.66 KiB)
```

The most interesting part of this is the call to `CUDA.@sync`. The CPU can assign jobs to the GPU and then doing other stuff (such as assigning more jobs to the GPU) while the GPU completes its tasks. Wrapping the execution in a `CUDA.@sync` block will make the CPU block until the queued GPU tasks are done, similar to how `Base.@sync` waits for distributed CPU tasks. Without such synchronization, you'd be measuring the time takes to launch the computation, not the time to perform the computation. But most of the time you don't need to synchronize explicitly: many operations, like copying memory from the GPU to the CPU, implicitly synchronize execution.

For that particular hardware used, the GPU computation was significantly faster than the single-threaded CPU computation, and the use of multiple CPU threads makes the CPU implementation competitive.

The high-level functionality of CUDA often means that you don't need to worry about writing kernels at a low level. However, there are many cases where computations can be optimized using low-level manipulations [**?**].

1.6 Modules and packages

A Julia `module` is a named sequence of `julia` code, typically contained in a file with the the same name:

```
module NameOfModule; <some code>; end # => Main.NameOfModule
```

A module start with the reserved word `module` followed by <*NameOfModule*> and is terminated by `end` word. Modules in Julia help organize code into coherent units and have the following features.

Modules are *separate namespaces*, each introducing a new *global scope*. This is useful, because it allows the same name to be used for different functions or global variables without conflict, as long as they are in separate modules.

Modules have facilities for detailed `namespace` *management*: each defines a set of names it `export`s, and can `import` names from other modules with `using` and `import`. Modules can be precompiled for faster loading, and may contain code for *runtime initialization*. Typically, in larger Julia packages you will see module code organized into program files [**?**]. One can have multiple files per module, and multiple modules per file. include behaves as if the contents of the source file were evaluated in the global scope of the including module.

The recommended style is not to indent the body of the module, since that would typically lead to whole files being indented. Also, it is common to use `UpperCamelCase` for module names (just like types),

A *software library* is a suite of data and programming code that is used to develop software programs and applications. In Julia a library is called *package*. A Julia package contains modules, tests, and documentation. It extends core Julia functionality. You can share your code with the community by developing a package. You can create a Julia package using as support the built-in package manager `PkgDev.jl` or the package `PkgTemplates.jl`. The second is easier for a novice.

A good tutorial about how a user may develop a package is [**?**]. The reader is warmly solicited to see, and try the suggested step-by-step development.

The Julia ecosystem contains over 9,000 packages that are registered in the `General registry`, which means that finding the right package can be a challenge. Fortunately, there are services that can help navigate the ecosystem, including:

1. `JuliaHub`: service that includes search of all registered open source package documentation, code search, and navigation by tags/keywords.
2. Julia `Packages`: to browse Julia packages, filter by categories, and sort them by popularity, creation date or date of last update. Also supports browsing package

Chapter 2
The Package Plasm.jl

`Plasm` is a software package for advanced geometry programming in Julia. In particular, it is oriented to generate engineering and architecture models whose computer representations may be classified as maps of topological polyhedra. This is a general domain of shapes, including linearized approximations of curved objects and, of course, the cellular complexes of simplicial, cubical, and polyhedral cells. Summing up, `Plasm.jl` is an open-source porting to Julia of the original `PLaSM`, an extension to the geometry of `FL` language at Function Level by Backus and his group at IBM Almaden in the 1990s. Written originally in Common Lisp, it was then ported to Scheme and C++, then to Python, and now to Julia.

2.1 Backus' functional programming

John W. Backus directed the IBM team that invented and implemented the FORTRAN (Formula Translation) language. FORTRAN was the first high-level scientific and technical computer language developed by IBM in 1954-1956 for scientific and engineering applications and subsequently came to dominate scientific computing. FORTRAN is used up to now on supercomputers.

FP (Functional Programming) and FL (Function Level)

The 1977 ACM Turing Award, in honor and recognition of Turing's contribution to the field of computing, was presented to John Backus at the ACM Annual Conference in Seattle. In introducing the recipient, Jean E. Sammet, Chairman of the Awards Committee, said that it was for Fortran and the BNF (Backus Normal Form) that he was receiving that year's Tur-

ing Award. Therefore, everybody was expecting Backus would describe the work done to implement FORTRAN at IBM in the 1950s.
Conversely, the Backus' Turing Lecture was entitled "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs" [?]. This lecture was enormously influential, and opened a decade and half of renewed research work about functional languages.

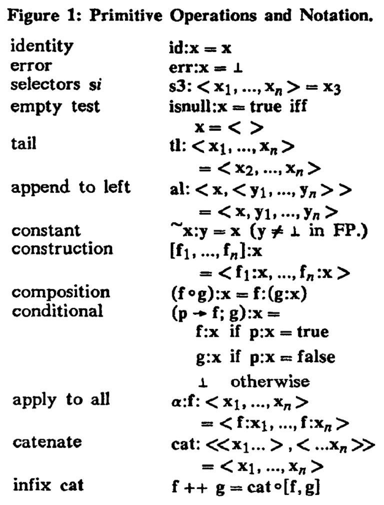


Figure 1: Primitive Operations and Notation.

identity	id:x = x
error	err:x = ⊥
selectors *si*	s3: $< x_1, ..., x_n > = x_3$
empty test	isnull:x = true iff x = < >
tail	tl: $< x_1, ..., x_n > = < x_2, ..., x_n >$
append to left	al: $< x, < y_1, ..., y_n > > = < x, y_1, ..., y_n >$
constant	$\tilde{}$x:y = x (y ≠ ⊥ in FP.)
construction	$[f_1, ..., f_n]$:x $= < f_1$:x, ..., f_n:x >
composition	(f ∘ g):x = f:(g:x)
conditional	(p → f; g):x = f:x if p:x = true; g:x if p:x = false; ⊥ otherwise
apply to all	α:f: $< x_1, ..., x_n > = <$ f:x_1, ..., f:$x_n >$
catenate	cat: $<< x_1 ... >, < ... x_n >> = < x_1, ..., x_n >$
infix cat	f ++ g = cat∘[f, g]

Fig. 2.1 The primitive operations (combining forms) and notations [?] of FP programs. The semantics of FP is embodied in an underlying algebra of programs, a set of function level equalities that may be used to transform programs and to reason about them.

A simple *functional programming* (FP) system is described in [?]. It is based on combining forms for building programs from simpler programs. An *algebra of programs* is described whose variables denote FP functional programs and whose "operations" are FP functional forms, the combining forms of FP programs, to support the function-level programming paradigm. It allows building programs from a set of generally useful primitives and avoiding named variables. Only programs, aka functions, may be named. In Figure 2.4 we display the algebraic rules of FP.
FL (short for "Function Level") is a programming language created by IBM Research at Almaden. FL is the result of the effort [?] to design a practical functional programming language based on Backus' FP. FL is intended to be a programming language in which it is easy to write clear, concise, and efficient programs. FL is designed around a rich set of functionals, forms for combining existing programs to construct new ones. This emphasis on programming at the function level results in programs that have a rich mathematical structure useful in reasoning about

and optimizing programs [?]. A short description of FL combining forms and primitive functions is given below, where its geometric extension with Julia's functional syntax is discussed.

2.2 FL-based PLaSM in Julia syntax

Here we gives a brief informal description of the `Plasm` language [?, ?], together with few examples. The section is intended to acquaint the reader with the syntax and style of `Plasm` programs without going into details.
The interpreter and interactive GUI for PLASM were developed at the Sapienza University of Rome in the 1990s, extending the FL syntax and semantics with the only addition of a geometric type called HPC (hierarchical polyhedral complex) and supporting the FL programming style for geometric computing within the wide domain of Computer-Aided Design.
The book GP4CAD (Geometric programming for CAD) [?] provides some hundreds of small, simple programs using the FL syntax.
In the past decade, the command-line user interface (CLI), the geometric computational kernel, and the interactive visualizer of "classic" `Plasm` were ported to Python [?] and C++, respectively, mainly using the functional features of such languages [?]. Analogously, in recent times, a native and extended port to Julia is being carried out [?], used in this book, and briefly described in the following, abridged in a few general points.

a. Function *application* and binary function *composition* are native in Julia, which provides `f(`x`)` and `g ∘ f`, respectively.
b. FL *sequence* `<`$x_1, x_2, \dots, x_n$`>` is implemented as array `[`$x_1, x_2, \dots, x_n$`]`.
c. All primitive functions (programs) are *pure* (without side effects) and written *uppercase*, while named expressions have capitalized names.
d. Each `Plasm` function is *unary*, possibly using an array of arguments.
e. The geometric types `Hpc` and `Lar` are extended with an optional Julia's dictionary of properties.

The point *d.* is sometime relaxed in Julia `Plasm`, to reduce the visual rumor. From [?], where the interested reader may find many codes discussed in this book in their FL version, we recall that significant advantages are obtained with this approach in the style and efficiency of program development. More generally, it is well known that functional programming enjoys several good properties:

- The set of syntax rules of a functional language is very small.
- Each rule is very simple.
- The program code is terse and clear.
- The meaning of a program is well understood, since there is no state.
- Functions may be used both as programs and as data.

- Programs are easily connected by concatenation and nesting.

Programs are functions

Generally speaking, a `Plasm` program is a *function*. When *applied* to some input *argument*, a program produces some output *value*. Two programs are usually connected by using functional composition, so that the output of the first program is used as input to the second program. Starting from here we use the Julia syntax.

Program composition and application

The composition of `Plasm` functions, i.e., `FL` programs with Julia syntax, works exactly as the composition of mathematical functions. E.g., the application of the composite mathematical function $f \circ g$ to the x argument

$$(f \circ g)(x) \equiv f(g(x))$$

means that the function g is first applied to x and that the function f is then applied to the value $g(x)$. The Julia `Plasm` notation for the previous expression is:

```
(f∘g)(x) ≡ f(g(x))
```

where ∘ stands for binary function *composition* and `g(x)` stands for *application* of the function `g` to the argument `x`. Both notations are Julia's native.

Naming objects

In `Plasm`, a name can be assigned to every value generated by the language, by using an (unmutable) *definition* construct, either with or without explicit parameters. In both cases the so-called *body* of the definition, i.e. the expression which follows the definition *head*, at the right hand of the "`=`" symbol, will describe the computational process which generates the *value* produced by the computation. The parameters which it implicitly/explicitly depends on may be embedded in such a definition. For example, we may have:

```
object = (Fun3 ∘ Fun2 ∘ Fun1)(params);
```

The computational process which produces the `object` value can be thought as the computational pipeline shown in Figure 2.2.
In this example, the reliance of the model on the parameters is implicit. To modify the generated object value it is necessary (a) to change the source code in either the body or the local environment of its generating

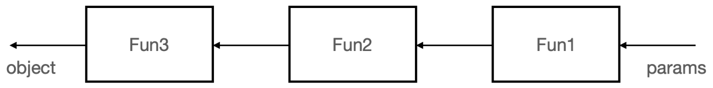


Fig. 2.2 Example of computational pipeline

function; (b) to compile the new definition; and (c) to evaluate again the object identifier.

Parametrized objects

A *parametric* geometric model can be defined, and easily combined with other such models, by using a generating function with formal *parameters*. Such kind of function may be instanciated with different actual *arguments*, thus obtaining different output values. For example, we may have:

```
object(params) = (Fun3 ∘ Fun2 ∘ Fun1)(params);

obj1 = object([p₁, p₂, … , pₙ];
obj2 = object([q₁, q₂, … , qₙ];
```

It is interesting to note that the generating function of a geometric model may accept parameters of *any* type, including other geometric objects.

2.3 Geometric Programming at Function Level

As we already know, `Plasm` is a geometry-oriented extension of a subset of the `FL` language [**?**, **?**], which is a pure functional language based on combinatory logic [Wikipedia]. In particular, the `FL` language makes use of both pre-defined and user-defined *combinators*, i.e. higher-order functions which are applied to functions to produce new functions. The small but very significant `FL` subset which is used as the base environment of `Plasm` is summarized in this section.
Notice that here and in the remainder of this book the infix symbol ≡ is normally used to tell the reader that the *expression* on its left side evaluates to the *value* on its right side. Sometimes this symbol is also used to denote an equivalence between syntactical forms.

Elements of FL syntax in Julia

Primitive `FL` *objects* are characters, numbers and truth values. Primitive objects added to it by `Plasm` are `Hpc` and `Lar` geometric values, discussed in following chapters. Primitive objects, functions, applications and sequences are *expressions*. *Sequences* are expressions separated by commas and contained within a pair of square brackets (Julia `Vector` type):

```
[5, fun]
```

An *application* expression `exp1(exp2)` applies the *function* resulting from the evaluation of `exp1` on the *argument* resulting from the evaluation of `exp2`. Julia allows some binary functions to be used both in infix and prefix form:

```
1 + 3 ≡ +(1,3) ≡ 4
```

Application associates to left, i.e. a sequence of repeated applications is evaluated from left to right. Note that this is only possible if all the applications, but possibly the last one, generate a new function to be applied to the next argument:

```
f(g)(h) ≡ (f(g))(h)
```

Application binds stronger than composition, i.e. applications are evaluated first before compositions, as it is shown in the following example. Of course the application of `f` must generate a function value:

```
f(g) ∘ h ≡ (f(g) ∘ h)
```

Combining forms and functions

The function level approach to programming of `FL` emphasizes the definition of new functions by combining existing functions in various ways. The result of this approach is a programming style based on function-valued expressions. Some important `FL` *combining forms* and functions follow.

Construction

The combining form `CONS` allows application of a sequence of functions to an argument, so producing a sequence of applications:

$$\texttt{CONS([}f_1,\ldots,f_n\texttt{])(x)} \equiv [f_1(x),\ldots,f_n(x)]$$

A `CONS`ed sequence of functions is a sort of *vector function*, that can be composed with other functions and that can be applied to data.
E.g., `CONS([sin,cos,tan,atan])` when applied to the argument `π` returns the sequence of applications

```
CONS([sin,cos,tan,atan])(π)     #=
4-element Vector{Float64}:
  0.0
 -1.0
  0.0
  1.2626272556789115    =#
```

Apply-to-all

The combining form `AA` has a symmetric effect, i.e. it applies a function to a sequence of arguments giving a sequence of applications. It is equivalent to the functional `map` of other languages, Julia included:

```
AA(f)([x₁,...,xₙ]) ≡ map(f, [x₁,...,xₙ]) ≡ [f(x₁),...,f(xₙ)]
```

For example, we may apply the trigonometric `SIN` function to all the elements of a list of numeric expressions:

```
AA(SIN)([0, π/3, π/6, π/2])
≡ [SIN(0), SIN(π/3), SIN(π/6), SIN(π/2)]
≡ [0, 0.8660254037844382, 0.49999999999999956, 1.0];
```

The reader should notice that numeric computations often introduce round-off and approximation errors. Just remember that π is an irrational number and cannot be represented exactly by using finite precision arithmetic. Also, functions like `SIN` are computed by using some truncated series expansion.

Identity

The function returns its argument unchanged

```
ID(x) ≡ x
```

In other words, the application of the identity function to *any* argument, gives back the same argument:

```
ID(0.5) ≡ 0.5
ID(SIN) ≡ SIN
ID(SIN)(0) ≡ SIN(0) ≡ 0
```

Constant

The combining form `K` is evaluated as follows, for whatever x_1 and x_2:

```
K(x₁)(x₂) ≡ x₁
```

In other words, the first application returns a constant function of value x_1, i.e. such that when applied to *any* argument x_2, *always* returns x_1. Some concrete examples follow:

```
K(0.5) ≡ Anonymous-Function
K(0.5)(10) ≡ 0.5
K(0.5)(100) ≡ 0.5
K(0.5)(SIN) ≡ 0.5
```

Composition

The binary composition of functions `COMP`, denoted in Julia by the symbol "`∘`", is defined in the standard mathematical way, as we already know:

```
(f ∘ g)(x) ≡ f(g(x))
```

where `∘` is obtained via the LaTeX expression `\circ` followed by `TAB` character. This important typing mechanism is standard in Julia and allows the program code to use Greek letters and many mathematical symbols. The *n-ary composition* of functions is also allowed:

```
COMP([f, g, h])(x) ≡ (f ∘ g ∘ h)(x) ≡ f(g(h(x)))
```

In the following we have, where `π`, `COS` and `ACOS` are the `Plasm` denotations for Julia's π, `cos` and `acos` [1], respectively:

```
(ACOS ∘ COS)(π) ≡ ACOS(COS(π)) ≡ ACOS(-1) ≡ 3.141592653589793
(COS ∘ ACOS)(-1) ≡ COS(ACOS(-1)) ≡ COS(3.141592653589793) ≡ -
    1
COMP([acos, cos, acos])(-1) ≡ ACOS(COS(ACOS(-1))) ≡
    3.141592653589793
```

Conditional combinator

This combinator has the following semantic: "`IF` the predicate `p` applied to object `x` is `true`, `THEN` apply `f` to `x`; `ELSE` apply `g` to `x`". This construct is very useful when it is necessary to apply different actions to input data depending on the value of some predicate evaluated on them, and is possibly more "natural" than the conditional statements available in other languages.
Formally, the conditional form `IF([ p, f, g ])` is evaluated as follows:

[1] Which can be directly used in the `PLASM` code, of course.

```
IF([ p, f, g ])(x)
    ≡ f(x) if p(x) ≡ TRUE
    ≡ g(x) if p(x) ≡ FALSE
```

From a syntax viewpoint, we remark that the `IF` operator is a higher-order function that *must* be applied to a *triplet of functions* in order to return a function which is in turn applied to the input data.
A *predicate* is a function `p: T →` $\{true, false\}$ `where T is a Type`. Both `true` and `false` are called *truth values*, and in Julia are `Bool` values. The predicate `p` is a *function*, as well `f` and `g`, to be alternatively executed depending on the truth value of the logical expression `p(x)`. E.g., we have:

```
    IF([ISINTPOS, K(true), K(false)])(1000) ≡ true
    IF([ISINTPOS, K(true), K(false)])(-1000) ≡ false
```

where `ISINTPOS` is a predefined predicate that returns `true` when applied to some positive integer.

Insert Right/Left

The combining forms `INSR` and `INSL` allow the user to apply a *binary* function `f`, with signature[2] `f: D × D → D`, on a sequence of arguments of *any* length n. In other words, implicitly it is: `INSR(f):` $D^n \to D$. Note that in the right-hand expressions below, `f` is always applied esplicitly to a *pair* of arguments:

```
INSR(f)([x₁, x₂,…, xₙ]) ≡ f([x₁, INSR(f)([x₂,…, xₙ])])
INSL(f)([x₁,…, xₙ₋₁, xₙ]) ≡ f([INSL(f)([x₁,…, xₙ₋₁]), xₙ])
```

An interesting use example of the `INSL` combinator is given below, where the function `BIGGER : Num × Num → Num` is defined. The `BIGGER` function returns the maximum of *two* arguments; the `BIGGEST:` $\texttt{Num}^n \to$ `Num` does the same from a list of arguments of *arbitrary length*:

```
BIGGER   # predefined function
BIGGEST  = INSL(BIGGER)
SMALLER  # predefined function
SMALLEST = INSL(SMALLER)

BIGGER([-10, 0]) ≡ 0
BIGGEST([-10, 0, -100, 4, 22, -3, 88, 11]) ≡ 88
```

[2] The *signature* of a function f from a *domain* A to a *codomain* B is the ordered pair of sets (A, B). It is normally associated to f by writing $f : A \to B$.

Catenate

The CAT function appends to the first one any number of input sequences, so creating a single output sequence:

```
CAT([[10,30,20],[11],[-7,8,12]]) ≡ [10,30,20,11,-7,8,12])
```

A pair of concrete examples of how the CAT function is used follows. The second one is quite interesting: it gives a *filter* function used to select the non-negative elements of a number sequence:

```
CAT([[10,30,20],[11],[-7,8,12]]) == [10,30,20,11,-7,8,12]
    #=
true    =#
(CAT ∘ AA(IF([ LT(0), K([]), ID ])))([-101,23,-37.02,0.1,84])
≡ CAT([ [], [23], [], [0.1], [84] ])
≡ [23, 0.1, 84]
```

It is useful to *abstract* a filter function with respect to a predicate and to an argument sequence, by showing this function semantics where curried LE stands for *less or equal* to its first argument. LT, GE, GT are similar.

```
FILTER(predicate)(sequence)
FILTER(LE(0))([-1,0,1,2,3,4]) == [-1, 0] # => true
```

Distribute Right/Left

The functions DISTR and DISTL are defined as:

```
DISTR([[a,b,c], x]) ≡ [[a,x], [b,x], [c,x]]
DISTL([x, [a,b,c]]) ≡ [[x,a], [x,b], [x,c]]
```

They accordingly transform a *pair*, constituted by an arbitrary expression and by an arbitrary sequence, into a *sequence of pairs*.

Script 2.3.1

Let us give an example of Plasm use. The Euler number e is defined as the sum of a series of numbers. In particular:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots$$

We compute an *approximation* of e, named euler, as the sum of the first 21 terms of the series. The factorial function is native in Julia:

```
  euler = (ADD ∘ AA(DIV) ∘ DISTL)([1, AA(factorial)(0:20)])
```

The number 20 is the highest positive integer for which the expression `factorial(20)` does not overflow out of memory assigned to an `Int64` number. In Julia, you can use a type `BigInt` with the `big` function, converting a number to a maximum precision representation. Hence, redefine `factorial` as the `FACT` function below, which uses the *ternary conditional* statement:

```
FACT(n) = n>0 ? *(1:big(n)...) : 1  #=
Fact (generic function with 1 method)   =#
FACT(100) #=
9332621544394415268169923885626670049071596826438162146859296
3895217599993229915608941463976156518286253697920827223758 2
511852109168640000000000000000000000000  =#
```

Coding 2.3.1 (Computation by substitution) The `euler` value is computed here by successive substitutions. Of course, the Julia's optimizing compiler might do a much better job:

```
euler = (ADD ∘ AA(DIV) ∘ DISTL)([1,AA(FACT)(0:9)])
≡ (ADD ∘ AA(DIV) ∘ DISTL)([1, AA(FACT)([0, 1, 2,…, 8, 9])])
≡ (ADD ∘ AA(DIV) ∘ DISTL)([1, [FACT(0),FACT(1),…,FACT(9)]])
≡ (ADD ∘ AA(DIV) ∘ DISTL)([1, [1,1,2,6,…,40320,362880]])
≡ (ADD ∘ AA(DIV)(DISTL([1, [1,1,2,6,…,40320,362880]]))
≡ (ADD ∘ AA(DIV)([[1,1], [1,1], [1,2], [1,6],…,[1,362880]])
≡ ADD([ 1/1, 1/1, 1/2, 1/6, …, 1/40320, 1/362880 ])
≡ 2.7182815255731922
```

Above, we have seen our first substantial example of `Plasm`, aka `FL` computation, as a sequence of expression transformation using the rules of combinators. The round brackets induce the order of transformations included into an expression and often corresponds to applications. The sub-expressions nested more deeply are transformed first. When using `Float64`, i.e., 8 bytes, the numeric precision is 15-16 digits.

A simpler and more elegant implementation of the Euler number is given below, where `C` is the currying combinator[3]:

```
    EULER(n) = (ADD ∘ AA(C(DIV)(1) ∘ FACT))(0:n)
```

The best Julia approximation of the Euler number is with `n = 57` terms of the defining series, since all digits (80) of a `BigFloat` value are exact:

```
EULER(56)   #=
2.718281828459045235360287471352662497757496954162242915473 4
```

[3] In mathematics and computer science, *currying* is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

```
483565013216246202514   =#
EULER(57)   #=
2.7182818284590452353602874713526624977574969541622429154734
483565013216246202549   =#
EULER(58)   #= The 58th element of the Euler series
2.71828182845904523536028747135266249775724969541622429154734
483565013216246202549   =#
```

2.4 Julia's package Plasm.jl

After several years of research about Linear Algebraic Representation (LAR) and algebraic operations [**?**, **?**, **?**, **?**] with sparse matrices and solid models, a new Julia package for geometric programming, named Plasm.jl, was developed while writing this book. We aim to finish the software version 1.0 before the book is published.

The work has mainly consisted of porting to Julia the Pyplasm library [**?**] written in Python years ago. Of course, both are open-source and downloadable from the web[4]. Our software plan is to realize in version 1.0 a significant extension of the language related to computation of space arrangements [**?**] and Boolean solid algebras [**?**].

Of course, the reader is warmly invited to install the latest version of Julia and to download our package `Plasm.jl` on computational environment. The installation is not strictly necessary since web access will be available. Still, it is always helpful, while learning a new language, to have your own environment where you are free to do any experiment.

Plasm.jl

The main file of the package is named as usual with the name `Plasm.jl` of the package itself. Its primary function is to create the run-time executable of `Plasm.jl`, by calling the external references (i.e., the exported functionality) taken from other packages and to include the Julia code of other package files, in our case `fenvs.jl`, `hpc.jl`, and `viever.jl`. Note that its name starts uppercase, according to the Julia's convention for packages.

fenvs.jl

The `fenvs.jl` file, whose name stands for "`f`unctional `env`ironment`s`" implements in Julia the majority of the small exciting programs developed for the GP4CAD book and includes many primitives for surface design with various

[4] Citation of pyplasm and plasm.jl URLs.

methods. Other needed functions and functions of general use can be created directly by the reader if savant in computer graphics or CAD.

hpc.jl

The primary data structure is the `Hpc` (Hierarchical Polyhedral Complex) [**?**], based on convex cells, and extended to allocate general polyhedral cells, i.e., polyhedra of any (low) dimension, connected but generally nonconvex and with interior holes. The `hpc.jl` file contains the developed design and implementation of the very general and multidimensional geometric data structure called `Hpc` [**?**], that allows to accommodate all the geometric and solid algorithms and tools developed in this book. As we show in the book, the `Hpc` structure is straightforward, versatile, and general. It does not use any of the large number of highly specialized and very complex data structures invented for solid modeling. We will show that our topological approach to geometric computations only needs the linear algebra of sparse matrices and vectors.

viever.jl

Finally, the `viewer.jl` file is used as the home of an interactive geometry viewer, used to interact with the shapes generated by Plasm codes either on the terminal screen of a laptop or on the HTML interface of a web browser program. This web viewer version was developed to view the Plasm models on the web and to write rich text examples and exercises within the pages of a web notebook, and hence without the need to install any software.

2.5 Julia REPL (Read-Eval-Print-Loop)

An interactive language shell is an interactive computer programming environment in a single terminal that takes user inputs from keyboard or file, executes them, and prints the result to the display. A program developed in REPL terminal environment (CLI – command language interface) is written and executed piecewise. This approach requires that the language executable code may work as an interpreter, i.e. combining translation of source lines in machine code with immediate execution.

REPL editor

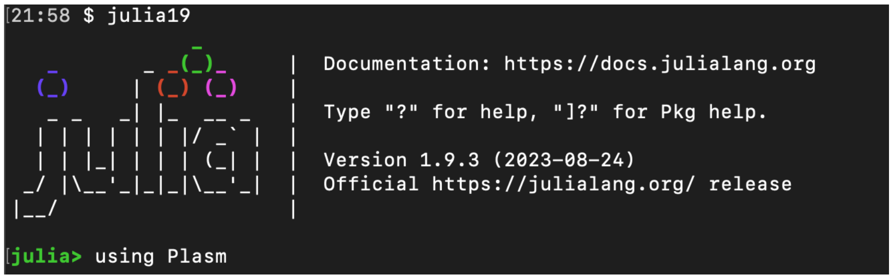
Julia comes with a full-featured interactive command-line REPL (read-eval-print loop) built into the julia executable [**?**]. Once installed, to enter Julia, open a terminal and digit ”julia” after the system prompt “$”. The acknowledgement in Figure 2.3 will appear on the screen. Once the REPL starts, you will be at the Julia prompt. The Julia REPL can operate in different prompt modes:

- Julia mode (the default),
- help mode,
- Pkg mode, and
- shell mode.

To enter help, Pkg, or shell mode, place the cursor at the beginning of the Julia mode prompt and type a question mark (?), a closing bracket (]), or a semicolon (;), respectively.

The help mode is used to get information about the meaning or use (arguments, examples, etc.) of a function, as written by the developer in the function "docstring". The Pkg (package) mode covers many things, including managing package installations, developing packages, working with package registries and more. The shell mode allows the terminal user to run shell commands from the Julia `REPL`.

To return to Julia mode, place the cursor at the beginning of the prompt and press Backspace [**?**]. A simple and very useful interactive blog to start use `REPL` is https://blog.glcs.io/julia-replheading-starting-the-julia-repl.



```
[21:58 $ julia19
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.9.3 (2023-08-24)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

[julia> using Plasm
```

Fig. 2.3 To run Julia by invoking its name on the commend-line of the terminal on computer screen. Note the "using Plasm" after the REPL prompt "> julia".

OhMyREPL package

The Julia package `OhMyREPL` hooks into the Julia `REPL` and gives it many new features [**?**]. It allows for several enhancements, including: syntax highlighting; bracket highlighting; bracket completion; rainbow brackets; markdown syntax highlighting; fuzzy `REPL` history search.

Julia REPL workflow

The most basic Julia workflows involve using a text editor in conjunction with the julia command line. A common pattern includes the following elements:

1. Put code under development in a temporary module. Create a file, say `Tmp.jl`, and `include` within it

```
    module Tmp
    <your definitions here>
    end
```

2. Put your test code in another file. Create another file, say `tst.jl`, which begins with

```
    import Tmp
```

   and includes tests for the contents of `Tmp`. The value of using `import` versus `using` is that you can call reload(”Tmp”) instead of having to restart the REPL when your definitions change. Of course, the cost is the need to prepend `Tmp.` to uses of names defined in your module. (You can lower that cost by keeping your module name short.) Alternatively, you can wrap the contents of your test file in a module, as

```
    module Tst
        using Tmp
        <scratch work>
    end
```

3. The advantage is that you can now do `using Tmp` in your test code and can therefore avoid prepending Tmp. everywhere. The disadvantage is that code can no longer be selectively copied to the REPL without some tweaking.
   Lather. Rinse. Repeat. Explore ideas at the julia command prompt. Save good ideas in `tst.jl`. Occasionally restart the `REPL`, issuing

```
    include(“Tmp.jl")
    include("tst.jl")
```

Julia’s REPL provides rich functionality that facilitates an efficient interactive workflow [?].

2.6 Geometric Programming examples

Here we show a couple of simple examples to show the compactness and expressive power of our geometric language.

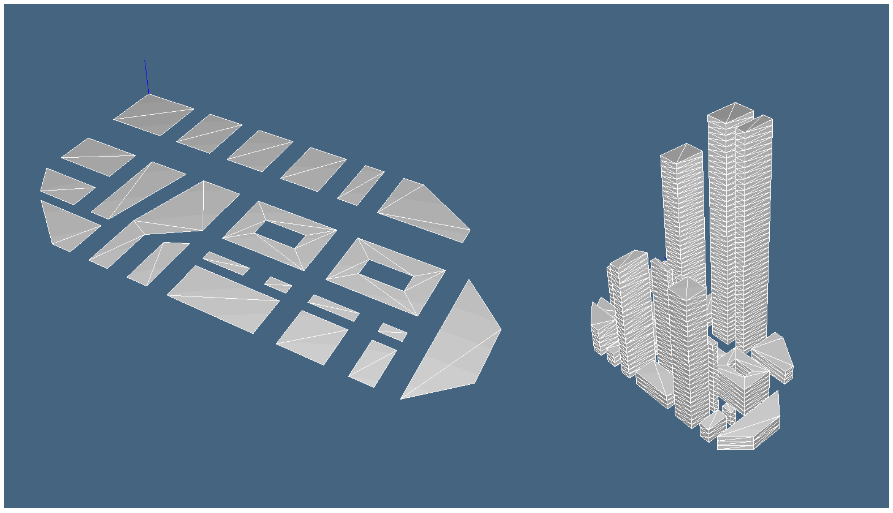


Fig. 2.4 (a) Hierarchical polyhedral complex (Lar) 2D value embedded in 3D. A perspective projection generates the image. Each connected shape is a polyhedral cell, even nonconvex and with inner holes. Each triangle is a convex cell; (b) 3D Plasm view of the model value of `Hpc` type discussed in Example 2.6.2

Coding 2.6.1 (2D virtual Manhattan) Of course, we start our geometric modeling session by telling the Julia compiler[5] to use the `Plasm` package. Then, we give the compiler the code and data that follows, possibly contained in a file `Manhattan2D.jl`.

```
julia> using Plasm
```

We start by defining the 2D coordinates of the vertices of our model.

```
julia> verts =
   [[0.,0],[3,0],[5,0],[7,0],[8,0],[9.5,1],[10,1.5],[0,3],
[3,3],[5,3],[7,3],[8,3],[9.5,3],[0,4],[3,4],[5,4],[9.5,4],
[12,4],[9.5,5],[10,5],[12,5],[0,6],[3,6],[5,6],[0,7],[3,7],
[5,7],[9.5,7],[12,7],[9.5,8],[12,8],[0,9],[3,9],[5,9],[8,
9],[9,9],[12,9],[0,10],[3,10],[5,10],[8,10],[9,10],[9.5,10],
[10,10],[12,10],[6,11],[7,11],[0,12],[3,12],[9,12],[9.5,12],
[0,13],[3,13],[6,13],[7,13],[9,13],[9.5,13],[0,14],[3,14],[5,
14],[8,14],[9,14],[9.5,14],[10,14],[12,14],[0,15],[3,15],[5,
15],[8,15],[0,16],[6,16],[7,16],[9,17],[9.5,17],[10,17],[12,
17],[6,18],[7,18],[9,18],[9.5,18],[10,18],[12,18],[2,19],[3,
19],[5,19],[8,19],[9,19],[9.5,19],[10,19],[12,19],[5,20],[12,
20],[7,22],[10,22],[9,6],[12,6],[9,15],[9.5,15],[10,15],[12,
15]]
```

[5] To load `Plasm` in Julia, open a terminal, start your `julia` application, and after the prompt `julia>` write `using Pkg; Pkg.add("Plasm ")`

Then we give the following description of convex cells of type `Cells = Vector{Vector{Int64}}`:

```
julia> cells =
    [[1,2,9,8],[3,4,11,10],[5,6,13,12],[14,15,23,22],[16,
17,19,24],[7,18,21,20],[25,26,33,32],[27,95,28,35,34],[95,
96,29,28],[30,31,37,36],[38,39,49,48],[40,41,47,46],[41,61,
55,47],[55,61,60,54],[54,60,40,46],[42,43,51,50],[44,45,65,
64],[52,53,59,58],[56,57,63,62],[66,67,84,83,70],[68,69,72,
71],[69,86,78,72],[78,86,85,77],[71,77,85,68],[97,98,74,
73],[99,100,76,75],[79,80,88,87],[81,82,90,89],[91,92,94,93]]
```

Finally, `verts` and `cells` are transformed in a geometric object of type `Hpc` by the function `MKPOL` and stored in the Julia variable named `model`.

```
julia> model = MKPOL(verts,cells)
# Hpc ... ...
```

A `model` image is generated by the `Plasm` viewer, within a system window named ”Manhattan2D”, for interaction with mouse and arrow buttons.

```
julia> VIEW( model, "Manhattan2D” )
```

From terminal we might write: $ `julia path/Manhattan2D.jl` □

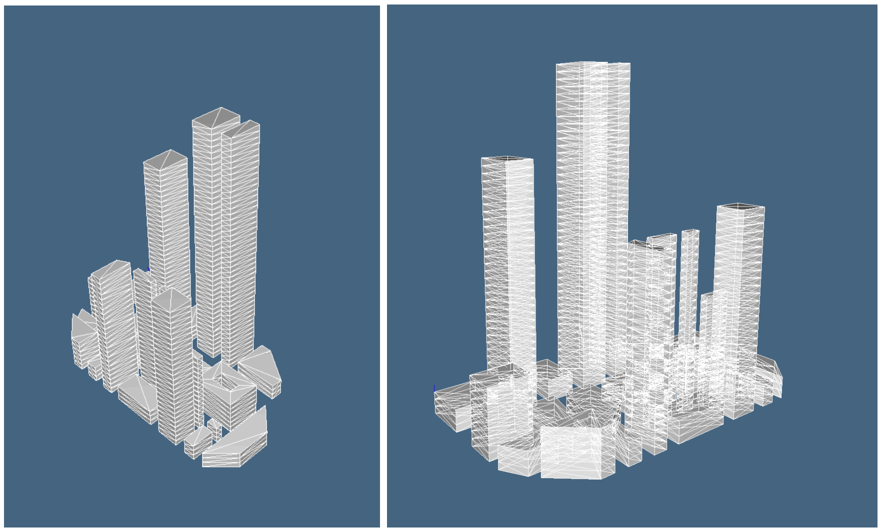


Fig. 2.5 3D model generated by using the 2D `cells` defined in `Manhattan2D` example. (a) opaque visualization; (b) transparent visualization from a different viewpoint.

Coding 2.6.2 (3D virtual Manhattan) The model of Figure 2.5 is given by (1) a vector `ManhattanH` of floor numbers; (2) generation of corresponding 1D and 2D vectors of `Hpc` objects stored in `pols1D` and `pols2D`; (3) by Cartesian product of corresponding `Hpc` pairs `(2D,1D)`. See below:

```
ManhattanH = [1,3,1,11,1,2,1,1,1,8,15,1,1,1,1,8,1,15,8,
    1,2,2,2,2,5,9,1,1,1].*3
# 29-element Vector{Int64}:
storeys = CONS(AA(DIESIS)(ManhattanH))(.5)
# 29-element Vector{Vector{Float64}}:
pols1D = AA(QUOTE)(storeys)
# 29-element Vector{Hpc}:
pols2D = [MKPOL(verts,[cell]) for cell in cells]
# 29-element Vector{Hpc}:
```

```
pols3D = AA(splat(*))(TRANS([pols2D, pols1D]))
# 29-element Vector{Hpc}:
VIEW(STRUCT(pols3D), "Manhattan3D")
```

In particular, we define an array of virtual heights for each *polygon* of `Manhattan2D`. Such numbers are transformed in repeated `storeys` heights by `AA(DIESIS)` (where `DIESIS` was the **#** operator in `FL`-based `Plasm`, but it is not usable in Julia, since it denotes comments) and then codified as 1D `Hpc` polyhedra stored in `pols1D` by the `QUOTE` operator. Analogously, a set of 2D polyhedra is stored in the `pols2D` array. The Cartesian product `*` of corresponding `Hpc` objects is stored in `pols3D` array using the Julia's `splat` function. Finally, a single `Hpc` object is visualized in a system window named `Manhattan3D`.

It is worth noting that every `polygon*segment` multiplication of two `Hpc` objects produces a new `Hpc` polyhedron of dimension equal to the sum of dimensions of operands. The reader should not forget that the `Plasm` language and the `Hpc` data structure are both *multidimensional*. □

We also note that a large number of significant programming examples with Julia and `Plasm` in Julia can be found inside the file `Plasm/fenvs.jl`, and executed in the terminal by writing $ `julia ./test/fenvs.jl`, being located into the `Plasm.jl directory`, and after having downloaded and installed the package `Plasm.jl` in a recent `julia` environment.

Coding 2.6.3 (Solid 2D graph of a scalar function) A simple unusual geometric example is given here, showing some of canonical constructs of geometric design with `Plasm`. First, we build and show in Figure 2.6a the `Domain2D` of the parametric 2D solid `model` given in Figure 2.6b. The `MAP` operator is applied first to the function `Mapping2D` to be applied to all *vertices* of a cell decomposition of `Domain2D`, in turn generated as the Cartesian product `POWER` of two 1D cellular complexes, so producing a 2D cell complex of the mapped interval `Domain2D` with 36×6 squared cells.

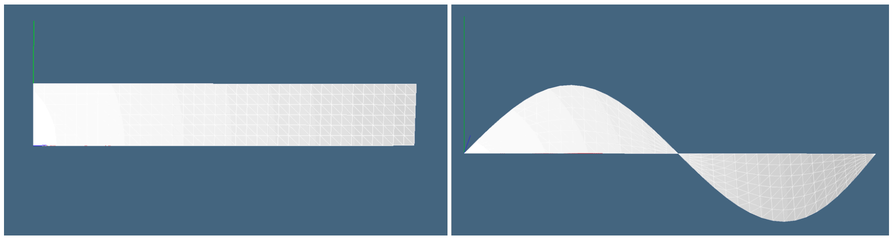


Fig. 2.6 The mapping of a function of two arguments on its 2D parameter domain. Each element (point and cell) of the cellular domain is paired with a corresponding element in the function range. The figure shows how the elements are paired. (a) The cellular complex decomposing the function domain $[2\pi, 1]$; (b) the function range transformed after the function `p->[u,sin(u)*v]` was mapped on the vertices of the 2D domain.

```
Domain2D = Power(INTERVALS(2π)(36), INTERVALS(1)(6));
Mapping2D = p->((u,v)=p; [u,sin(u)*v]);
model = MAP(Mapping2D)(Domain2D)
```

The two parameters `u,v` of the model domain are worked out by `p->(...)`, anonymous `Julia` function, and applied to each vertex of `Domain2D` by the `MAP` operator. Two geometric objects Domain2D and model are finally shown and displayed in Figure 2.6:

```
VIEW(Domain2D)
VIEW(model)
```

The virtual models of both `Domain2D` and `model`, finally visualized in Figure 2.6, are made available for display by the `Plasm` operator `VIEW`, and can be interactively handled by the user.

Chapter 3
Geometry and topology primer

It is implausible to set up a computer language and computational environment to define and deliver the geometric shape of construction elements and spaces without a pretty good understanding of geometric and topological concepts. Hence, this chapter introduces some basic notions about geometric spaces, operators, and properties, like linearity, that provide the computational foundation of shape definition and model construction. The essential elements of geometric spaces and cellular models, like affine transformations and simplicial, cubical, polyhedral, and chain complexes, constitute the substrate of `Plasm` discourse for building hierarchical assemblies of AEC products of any scale and complexity.

3.1 Geometric Spaces

In mathematics and science, a space is a set of objects with some added structure. In this section we discuss the geometrical spaces used to represent inside a computer memory the actual objects of natural, artificial or virtual environments, or better their geometric models, in order to study, visualize and/or simulate some associated physical behavior. In particular, we deal in this section with linear, affine, and convex spaces as sets of vectors and/or points with selected properties.

3.1.1 Vector space

The concept of vector space is undoubtedly the most helpful instrument of mind invented by mathematicians for scientists, engineers, and architects, to represent and study formally or graphically both the primary and complex arrangements and behaviors of our natural and artificial environment.

Definition 3.1 (Vector space) (or *linear space*) $\mathcal{V}$ over a field $\mathcal{F}$ is a set, closed w.r.t. two composition rules (the output of composition belongs to the set of inputs). □

The elements $v \in \mathcal{V}$ are called *vectors*, and are often represented by an oriented arrow with given direction, orientation, and length. The elements $\alpha \in \mathcal{F}$ are called *scalars*. The sum of two non-zero vectors $u, v \in \mathcal{V}$ is a third vector $w = u + v \in \mathcal{V}$ with direction and length different from both u and v. We may write $v + v = 2v$, and see here both the operations of a vector space: (a) addition of vectors, and (b) multiplication times a scalar.

The product of a vector by a scalar $\alpha\, v = v\, \alpha \in \mathcal{V}$ is a vector collinear with v and with different length if $\alpha \neq 1$. This explain the name "scalar" since a number "scales" (change the length of) the vector it multiplies.

The length of $u = \alpha\, v$ grows or shrinks w.r.t. v according to a positive $0 < \alpha < 1$. If $\alpha < 0$, then u has orientation opposite to v.

Linear independence

A *linear combination* of vectors is a new vector that is defined as a sum of scalar multiples of other vectors. Let $v_1, v_2, ..., v_n \in \mathcal{V}$ and $\alpha_1, \alpha_2, ..., \alpha_n \in \mathcal{F}$, with $\mathcal{V}$ a vector space on the field $\mathcal{F}$ of scalar numbers. The vector

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n = \sum_{i=1}^{n} \alpha_i v_i \in \mathcal{V}$$

is called a linear combination of vectors $v_1, v_2, ..., v_n$.

- Two or more vectors are said *linearly independent* if none of them can be written as a linear combination of the others;
- if at least one of them can be written as a linear combination of the others, then they are said *linearly dependent*.

Given a set of vectors, you can determine if they are linearly independent by writing the vectors as the columns of a matrix A, and solving $A\, x = 0$.

If there are non-zero solutions, then the vectors are linearly dependent. If the only solution is $x = 0$, then they are linearly independent.

The typical vector spaces in this book will be (a) the numeric field $\mathbb{R}$ for scalars and $\mathbb{R}^n$ for vector coordinates, with (b) dim $= d$, for $0 \leq d \leq 3$.

Examples

Coding 3.1.1 (Matrices $m \times n$ are vector spaces) . Generate a random 100×100 matrix `A = rand(100,100)`. The default of `rand()` function are values of type `Float64` in `[0.0,1.0]`, so we ask the compiler to compute the matrix multiplication by the scalar π, denoted by the Greek symbol. The generated value is shown commented and simplyfied for printing in small space:

```
A = rand(100,100) * π   #=
100x100 Matrix{Float64}:
 0.901069  …  0.172854
 0.357548     0.8537
 ⋮           ⋱
 0.519136     0.857736      =#
```

Coding 3.1.2 (Random vector generation.) Then, we generate a 100-vector of random integers within the interval `[0,100]`, by using the Julia `0:100` iterator:

```
b = rand(0:100, 100)    #=
100-element Vector{Int64}:
 61
 85
  ⋮
 51          =#
```

Coding 3.1.3 (Multiplication matrix-vector.) This operation is implemented natively and very efficiently for big matrices in Julia, as well the product and the sum of dense matrices (the detail of printing depends on available space on output screen):

```
A * b         #=
100-element Vector{Any}:
 2786.383421064984
 2554.1378635659034
    ⋮
 2735.1205994270754 =#
```

Coding 3.1.4 (Addition matrix-vector.) Conversely, the sum of a matrix with a vector is not a linear operation, so we need to broadcast (. operator) the vector `b` on all columns of `A`:

```
A .+ b        #=
100x100 Matrix{Float64}:
 61.9011  …  61.1729
 85.3575     85.8537
  ⋮         ⋱
 51.5191     51.8577    =#
```

Summing up: while the `Matrix` by `Vector` multiplication is a native operation on the linear space of `Number`s, The addition of `Matrix` and `Vector` is not, and the broadcast operator must be used. Same for the sum of a matrix and a single scalar value. The reader should try.

Subspace

Let V be a vector space on the field F. We say that $U \subset V$ is a *subspace* of V if U is a vector space with respect to the same operations. In particular, $U \subset V$ is a *subspace* of V if and only if:

1. $U \neq \emptyset$;
2. for each $\alpha \in F$ and $u_1, u_2 \in U$, $\alpha\, u_1 + u_2 \in U$

The *codimension* of a subspace $U \subset V$ is defined as $\dim V - \dim U$.

It may be useful to note that the intersection of subspaces is a subspace. In particular, if U_1, U_2 are subspaces of V, then $U_1 \cap U_2$ is a subspace of V.

Generators, Bases, and Coordinates

Span and generators

The smallest subset of vectors that can be generated by linear combinations of a subset $S \subset \mathcal{V}$ of linearly independent vectors is called *span* S.

The set S is therefore called a set of *generators*.

The *span* S is closed with respect to addition and multiplication, and hence is a *subspace* including the zero vector, which is contained in every subspace.

Bases and coordinates

The *basis* and *dimension* of the linear space $\mathcal{V}$ are a minimum set of generators for $\mathcal{V}$, and its number $d = \dim \mathcal{V}$ of elements, respectively.

Every basis of a linear space $\mathcal{V}$ has the same number d of elements.

When a basis for $\mathcal{V}$ has been fixed, i.e. an ordered minimal subset of generators $B \subset \mathcal{V}$ has been chosen, every vector $v \in \mathcal{V}$ can be expressed *uniquely* as the linear combination of elements of B with scalars. The ordered tuple of such scalars is called the *coordinate* tuple, or the coordinates, of vector $v \in \mathcal{V}$, and denoted as $[v]$.

Remark 3.1 It is to mention that to represent vectors by coordinates requires that a minimum set of space generators and their ordering have been already chosen. We will say that the space has been p*arameterized*, since every vector is *uniquely* identified by the linear combination of the basis elements with a unique tuple of scalars. □

Remark 3.2 The basis is often denoted by the ordered sequence $(e_1, \dots, e_d)$ of vector elements, or by the matrix $[e_1 \cdots e_d]$ of their coordinates by columns, where $e_i = [0, 0, 1, \dots, 0]^t$ is a (column) tuple of zeroes, with only one element 1 in position i. The *standard basis* of a coordinate vector space is the set of vectors whose components are all zero, except one that equals 1. □

Examples of vector spaces

The usual geometric example of vector space has the oriented arrows[1] as elements, summed with the parallelogram rule, and scaling related to elongation or shortening. Other examples are the linear spaces of matrices $\mathcal{M}_m^n(\mathbb{R})$ with real elements, m rows and n columns, and $\mathcal{M}_m(\mathbb{R})$ and $\mathcal{M}^n(\mathbb{R})$ of column and row vectors, respectively.

Linear space is also the space $\mathbb{P}_n(\mathbb{R})$ of real polynomial functions $p : \mathbb{R} \to \mathbb{R}$ such that $x \mapsto p^n(x)$ of degree $\leq n$. In particular, $p^n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n$ is exactly a linear combination of a tuple of $n+1$ scalars a_k, $0 \leq k \leq n$, with the *power basis* of polynomials x^k, $0 \leq k \leq n$.

The Bernstein bases of polynomial space $\mathbb{P}_n(\mathbb{R})$

In geometric modeling and computer graphics, the Bernstein-Bézier basis of polynomials is fundamental. The $n+1$ Bernstein polynomials of degree n, defined as

$$B_k^n(x) = \binom{n}{k} x^k \, (1-x)^{n-k}, \qquad 0 \leq k \leq n$$

form a basis for the vector space $\mathbb{P}_n(\mathbb{R})$ of polynomials of degree at most n with real coefficients. Therefore, to implement the basis and to draw a graph of each basis function we have to consider: the degree `n` we are interested to; the ordinal number `k` of each function ($1 \leq k \leq n$); and finally the independent variable x such that $x \mapsto p^n(x)$.

Coding 3.1.5 (Pure functional style in Julia) The julia function `B` given here returns the whole range of Bézier-Bernstein polynomials of any degree, and is very useful in curve geometric modeling.

```julia
B = n -> k -> u -> binomial(n,k) * u^k * (1-u)^(n-k)
```

This pure programming style is available in Julia and will be adopted when useful to get curried applications of functions. □

Let us make some checks: `B(1)(0)(0.5) == B(1)(1)(0.5) == 0.5 #=> true`, for the degree-1 basis made by polynomials `B(1)(0)` and `B(1)(1)`. The

[1] Formally: the equivalence classes of equipollent oriented arrows. In Euclidean geometry, equipollence is a binary relation between directed line segments.

vhole basis of degree n is generated by the higher-level function `Bernstein(n)` that we test in the quadratic case, where it returns a 3—element array of `Vector{Function}`:

Coding 3.1.6 (Generating Bernstein polynomial bases) The function `Bernstein(n)` is defined by mapping the partial function `B(n)` over the integer array `[0, 1, ..., n ]`.

```
Bernstein(n) = map(B(n), collect(0:n))
# => #7 Bernstein (generic function with 1 method)
Bernstein(2)
# => 3-element Vector{Function}
Bernstein(2)[2]
# => #9 (generic function with 1 method)
Bernstein(2)[2](0.1)
# => 0.18000000000000002
```

Now, we go to generate a discrete sequence of functions for the "vector" function `Bernstein(2)`, in order to test the implementation. Above we see also the second function of Bernstein basis of degree `n=2`, and finally its value `0.18` computed for `x = 0.1`.

Coding 3.1.7 (Sampling of third quadratic polynomial) The function `Bernstein(2)[3]` denotes the third function of the vector array `Bernstein(2)` of type `::Vector{Function}`, and we compute the three function values for `x=0.5`, that you can check in Figure 3.1.

```
Bernstein(2)[3] #=
37 (generic function with 1 method)    =#

Bernstein(2)[1](0.5) # => 0.25
Bernstein(2)[2](0.5) # => 0.5
Bernstein(2)[3](0.5) # => 0.25
```

Coding 3.1.8 (Sampling of third basis polynomial of degree 4) . We may look at the sequence of pairs $x, Bernstein(2)[3](x)$. A sampling of x values is created by `collect(0:0.1:1)`:

```
m = 10
for u in collect(0:1/m:1)
    println(" $(u), $(Bernstein(4)[3](u)) ")
end     #=
0.0, 0.0
0.01, 0.0005880600000000001
0.02, 0.00230496
0.03, 0.00508086
  ⋮     ⋮       =#
```

The Bernstein basis enjoy interesting properties. They are ≥ 0 for every value of the independent variable $x \in [0,1]$. Even more, for every x the elements of each basis sum to 1. Such bases are said *partition of unity*.

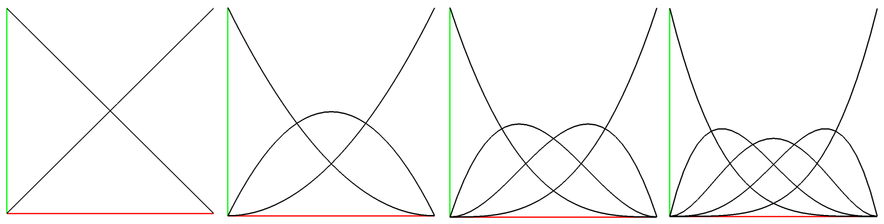


Fig. 3.1 The four images give the graph, in $[0,1] \times [0,1] \subset \mathbb{E}^2$, of Bernstein's basis of linear, quadratic, cubic, and quartic polynomials in linear spaces $\mathbb{P}_n(\mathbb{R})$, with $1 \leq n \leq 4$, and $\dim(\mathbb{P}_n(\mathbb{R})) = n + 1 = 2$, 3, 4, and 5, respectively.

Coding 3.1.9 (Example of* `FL` *&* `Plasm` *Combinators) We remark on the incredible expressive power of `FL` programming inherited in `Plasm.jl`.

It is greatly useful to analyze goals and types of subexpressions, recuperating backward the development process to write the result expression. First, we define the 1D model `dom1D` of the function domain $[0,1]$ with 36 intervals, which is a geometric value of `Hpc` type:

```
dom1D = INTERVALS(1.0)(36)      #=
Hpc(MatrixNd([[1.0, 0.0], [0.0, 1.0]]), BuildMkPol[BuildMkPol(
    points=[[0.0], [0.027777777777777776], ... =#
```

Then, consider the $\mathbb{E} \to \mathbb{E}^2$ vector function `CONS(...)∘S1` of one variable, that is applied to the vertices of `dom1D::Hpc` by the `MAP` operator.

The selector function `S1` is used to extract the first (and unique) coordinate value from the internal data structure `point` array. The `CONS` operator, acting on argument of `Vector{Function}` type, transforms a function array into a vectorial function of coordinate functions `x(u)`, `y(u)`, etc.

Hence, we have `F = CONS(::Vector{Function})∘Sd` : $\mathbb{E}^d \to \mathbb{E}^n$, where `d` is the number of coordinates of domain vertices (one in this case), and `n` is the dimension of the embedding space, i.e., the number of coordinate functions inside the array argument, i.e. `CONS([,])`, two in this case:

```
F(k) = CONS([ID, Bernstein(4)[k]])∘S1       #=
#122 (generic function with 1 method)   =#
```

Finally, we note that expressions like `MAP(F)(dom1D)` are used to apply the `F` vector function to an object of `Hpc` type, normally to bend it. Remember that we generated an array of `Hpc` curves, for `k=1:n+1` — in our case with `n=4`. For this purpose, we need to finally apply the `STRUCT` operator, to transform the array of curves into a single `Hpc` value.

```
STRUCT( MAP(F(k))(dom1D) for k=1:n+1 )
```

Change of basis

It is often necessary, given a basis in a linear space, to compute a novel parameterization of the space with respect to a different, possibly more simple or useful basis. In other words, may be necessary to compute new coordinates for the vectors of the space, with respect to a new basis. This operation is called *change of basis*.

For concreteness, let we have vector data in a 3D linear space, where we need to compute their representation in a different basis. Let us denote the new basis as (u_i), and the standard one as (e_i), with $i \in \{1,2,3\}$. We may write the change of coordinates as a matrix map $T : \mathcal{V} \to \mathcal{V}$, such that $[u_{ij}] \mapsto [e_{ij}]$, so transforming the old coordinates of the new basis into the standard basis.

Therefore we set $T\,[u_{ij}] = [e_{ij}]$ and, since it is $[e_{ij}] = \mathbb{I}_3$, i.e., the 3×3 identity matrix, the solution is $T = [u_{ij}]^{-1}$. Of course, this matrix exists provided that $\det [u_{ij}] \neq 0$. In other words, the vectors of the new basis must be linearly independent.

3.1.2 Affine space

In geometric modeling and computer graphics it is useful to distinguish between a space of vectors and a space of points (supported by vectors). A space of points, that provides for an operation of *displacement*, is called an *affine space*, and is represented here by the $\mathcal{A}$ symbol. In particular, we call *affine action* the function:

$$\mathcal{A} \times \mathcal{V} \to \mathcal{A},$$

so that the *displacement* from a point $a \in \mathcal{A}$ to a point $b \in \mathcal{A}$ is given. An affine space $\mathcal{A}$ is endowed with an operation of *difference* of points $\mathcal{A} \times \mathcal{A} \to \mathcal{V}$ where

$$a + v = b, \quad \text{and hence} \quad b - a = v \in \mathcal{V}$$

Remarks

We note that a **vector space** provides for an internal operation of (a) *sum* of two vectors and the external (b) *product* of a vector by a scalar, both returning a vector. Conversely, in an **affine space** we have an external operation of

(i) *difference* of points, returning a vector, and a (ii) sum of a point and a vector, called *displacement*, returning a point.

Even more, in a vector space $\mathcal{V}$ there is a distinguished element 0, the *zero vector*, which is contained in all subspaces. On the contrary, in an affine space $\mathcal{A}$ *all points are equivalent*, with no distinguished elements.

We usually say that a space of points has been parameterized when a *Cartesian system* (origin and basis) has been chosen. In order to use coordinates to make it easier to work with points we have to choose :

1. a point, called *origin*, to associate with the zero vector, and
2. an orthonormal *basis* of vectors.

The above steps consent to associate each point with the tuple of coordinates of its *displacement vector* from the origin.

In a vector space all the subspaces have at least a common element, the zero vector. Contrariwise, two affine subspaces may not have common elements. In such a case they are said *parallel*.

The dimension n of an affine space $\mathcal{A}$ is that of its supporting vector space $\mathcal{V}$. A common word for *affine subspaces* of d dimension is d-hyperplane, or d-hyperspace when $d > 3$. Lines and planes are affine subspaces of dimension 1 and 2, respectively.

Affine independence and local parameterization

In an affine space $\mathcal{A}$ of sufficiently high dimension n we say that two points are *affinely independent* when *non coincident*, three points when *non aligned*, four points when *non coplanar*, and so on.

In general, $d+1$ points ($d \leq n$) are affinely independent when the d vectors defined by the differences $p_k - p_0$ of points from one of them are linearly independent.

The affine independence of a subset of $d+1$ points is often used to establish *local coordinate systems* on lines, planes and higher dimensional subsets of points. Let us choose two non coincident points a, b on a 3D line. Any other point p of the line remains parameterized by the α scalar in the expression

$$p = a + \alpha(b - a).$$

Note that $b - a$ is a vector, $\alpha(b - a)$ is a vector, $a + \alpha(b - a)$ is a point plus a vector, which is a point. The typing of our expression looks correct!

Analogously, let us choose three non colinear points a, b, c in 3D space. They are certainly non coincident, and of course fix a plane, i.e., an unique affine subspace of dimension 2 embedded in space. A parameterization of this plane is given by the pairs (α, β) of scalars in the expression

$$q = \alpha(b - a) + \beta(c - a),$$

as the reader may immediately check. Similar local coordinates will hold in every affine d-subspace in n-dimensional space.

The typical affine space of points used in this book is the Euclidean space $\mathbb{E}^d$, $1 \leq d \leq 3$, usually furnished of a Cartesian system, with coordinates in `Float64`.

3.1.3 Convex space

Let us consider the Euclidean space $\mathbb{E}^d$, $d \in \{2,3\}$ affine over the reals, that is the fundamental space of geometry, intended to represent physical space.

Affine subspaces become *convex sets* when a numeric constraint is imposed on the possible parameter values of an affine combination of points. Two points $a, b \in \mathbb{A}$ become the extreme elements of a *line segment* $p = \alpha\, a + (1-\alpha)\, b$ as set of points, by adding the further constraint $\alpha + \beta = 1$ to the parameter values $\alpha, \beta \geq 0$, so posing $\beta = 1 - \alpha$. Analogously, setting $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \geq 0$ constraints the set of points combination of three non aligned points (a, b, c).

Positive, Affine and Convex Combination of Points

Let $p_1, p_2, \dots, p_n$ be affinely independent points in $\mathbb{E}^d$, and $\alpha_1, \alpha_2, \dots, \alpha_n$ be scalar in $\mathbb{R}$. Their combination $\alpha_1 p_1 + \alpha_2 p_2 + \cdots + \alpha_n p_n$ is said *positive*, *affine*, and *convex*, respectively, when

1. $\alpha_1, \alpha_2, \dots, \alpha_n \geq 0$ (positive combination)
2. $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1$ (affine combination)
3. $\alpha_1 + \alpha_2 + \cdots + \alpha_n = 1 \quad \text{and} \quad \alpha_1, \alpha_2, \dots, \alpha_n \geq 0$ (convex combination)

It may be interesting to verify that the affine combination of points is a point. Let us eliminate the α_1 scalar using the unitary sum of scalars constraint:

$$p = (1 - \alpha_2 - \cdots - \alpha_n) p_1 + \alpha_2 p_2 + \cdots + \alpha_n p_n \tag{3.1}$$
$$= p_1 + \alpha_2 (p_2 - p_1) + \cdots + \alpha_n (p_n - p_1) \tag{3.2}$$

which, of course, is a point in $\mathbb{E}^d$.

A convex combination is positive and affine.

The set of all convex combinations of points $C \subset \mathbb{E}^n$ is the *convex hull* of C. The convex hull is the smallest compact set containing the points in C. It is the intersection of all compact sets of $\mathbb{E}^d$ which contain $\operatorname{hull} C$.

Convex coordinates of a point $c \in \operatorname{hull} C$ are the scalars whose convex combination with C elements produces c. If $C \subset E^n$ has $n+1$ affinely independent elements, i.e., it is a simplex (see Section **??**) the convex coordinates of each $c \in \text{hull } C$ are *unique*.

Characterization of affine pyramids

Speaking of local parameterization of affine subspaces, we did not fix any constraint on the signs of the scalar parameters (usually in the field $\mathbb{R}$ of reals). If all the scalars are positive, then all the spanned points stay inside the interior of a planar (or solid) angle contained within the lower-dimensional affine subspaces generated. It would be not difficult to see that the whole subspace will be partitioned by an arrangement of subspaces centered on the fixed point of the set of linearly independent vectors. As an example, consider the partitioning of 3D space generated by the $8 = 2^3$ octants defined by 4 non coplanar points, and in particular those generated when three points are at unit distance by the first, and the three difference vectors are pairwise orthogonal.

Positive, Affine and Convex coordinates

Affine Coordinates for Convex Sets
 Barycentric Coordinates for Convex Sets
 Test for boundary points: interior and exterior

3.2 Cellular models

This section discusses computational topology concepts used to compute the 2D/3D space partition induced by a collection of geometric objects of dimension 1D/2D, respectively. Cellular models are are often called *meshes* in engineering and design jargon. A mesh is a representation of a domain by discrete cells. In many areas of geometric/numeric computational engineering, including geo-mapping, computer vision and graphics, finite element analysis, medical imaging, geometric design, and solid modeling, one has to compute incidences, adjacencies and ordering of mesh cells, generally using disparate and incompatible data structures and algorithms. Only sparse vectors and matrices are used in `Plasm` to compute the linear spaces of chains, called chain complex, from dimension zero to three.

Topological definitions

A *complex* is a graded set $S = \{S_i\}_{i \in I}$ *i.e.* a family of sets, indexed here over $I = \{0,1,2,3\}$. We use two different but intertwined types of complexes, and specifically complexes of *cells* and complexes of *chains*. Their definitions and some related concepts are given in this section. Greek letters are used for the cells of a space partition, and roman letters for chains of cells, coded as either (un)signed integers or sparse arrays of (un)signed integers.

Definition 3.2 (d-Manifold) A *manifold* is a topological space that resembles a flat space locally, *i.e.*, near every point. Each point of a d-dimensional manifold has a neighborhood that is homeomorphic[2] to $\mathbb{E}^d$, the Euclidean space of dimension d. Hence, this geometric object is often referred to as d-manifold. □

Homeomorphic neighborhood means "topologically equivalent", like a rubber patch, that can be stretched without changing its topology.

Definition 3.3 (Cell) A *p-cell σ* is a p-manifold with boundary ($0 \leq p \leq d$) which is piecewise-linear, connected, possibly non convex, and not necessarily contractible. This definition refers to cellular complexes used in this paper and is different from other ones because a cell is neither simplicial, nor convex, nor contractible. □

In `Plasm`, cells may contain internal holes; cells of CW-complexes [**?**] are, conversely, contractible to a point. We deal with *Piecewise-Linear* (PL) cells of dimensions 0, 1, 2, and 3, respectively. It should be noted that 2- and 3-cells may contain holes, while remaining connected. In other words, `Plasm` cells are p-polyhedra, *i.e.* segments, polygons and polyhedrons embedded in 2D or 3D space.

Definition 3.4 (Cellular complex) A *cellular p-complex* is a finite set of cells that have at most dimension p, together with all their r-dimensional boundary faces ($0 \leq r \leq p$). A *face* is an element of the PL boundary of a cell, that satisfy the *boundary compatibility* condition. Two p-cells α, β are said boundary-compatible when their point-set intersection contains the same r-faces ($0 \leq r \leq p$) for both α and β. □

A cellular p-complex is said *homogeneous* when each r-cell ($0 \leq r \leq p$) is face of a p-cell.

Definition 3.5 (Skeleton) The s-skeleton of a p-complex Λ_p ($s \leq p$) is the set $\Lambda_s \subseteq \Lambda_p$ of all r-cells ($r \leq s$) of Λ_p. Every skeleton of a regular complex is a regular subcomplex. The difference $\Lambda_r - \Lambda_{r-1}$ of two skeletons is the set U_r of r-cells. □

[2] Homeomorphic means opologically equivalent.

Definition 3.6 (Support space) The support space Λ of a cellular complex is the point-set union of its cells. □

Definition 3.7 (Characteristic function) Given a subset S of a larger set A, the characteristic function $\chi_A(S)$, also called the *indicator function*, is the function defined to be identically one on S, and zero elsewhere. [**?**]. □

3.2.1 Simplicial complex

Definition 3.8 (Join operation.) The *join* of two compact sets of points $P, Q \subset \mathbb{E}^n$ is the set PQ of convex combinations of points in P and in Q. The join operation is associative and commutative. □

Definition 3.9 (Simplex.) A *d-simplex* $\sigma_d \subset \mathbb{E}^n$ $(0 \leq d \leq n)$ may be defined as the repeated join of $d+1$ affinely independent points, called *vertices*.□

A d-simplex can be seen as a d-dimensional triangle: a 0-simplex is a *point*, a 1-simplex is a *segment*, a 2-simplex is a *triangle*, a 3-simplex is a *tetrahedron*, and so on.

Coding 3.2.1 (Plasm d-dimensional simplex) A d-simplex is generated by the `Plasm` package by the `SIMPLEX` multi-dimensional function:

```
julia> using Plasm

julia> SIMPLEX::Function
SIMPLEX (generic function with 1 method)
```

Coding 3.2.2 (Dataset associated to simplices.) The simplex datasets follow, for the first integer parameters. Note that `SIMPLEX(d)` contain $d+1$ coordinate points of dimension d, and that `hulls` fields contain $d+1$ indices of points, i.e. a single *convex* cell:

```
SIMPLEX(0)          #=
Hpc(MatrixNd(1), Geometry([Float64[]], hulls=[[1]])) =#

SIMPLEX(1)          #=
Hpc(MatrixNd(2), Geometry([[0.0], [1.0]], hulls=[[1, 2]])) =#

SIMPLEX(2)          #=
Hpc(MatrixNd(3), Geometry([[0.0, 0.0], [1.0, 0.0], [0.0, 1.0]],
    hulls=[[1, 2, 3]])) =#

SIMPLEX(3)          #=
Hpc(MatrixNd(4), Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4]])) =#
```

```
SIMPLEX(4)          #=
Hpc(MatrixNd(5), Geometry([[0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0,
    0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0,
   0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4, 5]])) =#
```

For the sake of visual simplicity, we remove the `julia>` prompt for Plasm scripts, and use the multilinear comment to reduce their visual rumor.

Definition 3.10 *Skeleton and Faces.* The set $\{v_0, v_1, \dots, v_d\}$ of vertices of σ_d is called the *0-skeleton* of σ_d. The s-simplex generated from *any* subset of $s+1$ vertices $(0 \le s \le n)$ of σ_d is called an s-*face* of σ_d. □

Remark 3.3 Let us notice, from the definition, that a simplex may be considered both as a purely *combinatorial object* and as a *geometric object*, i.e. as the compact point-set defined by the convex hull of a discrete set of points.□

Definition 3.11 (Simplicial Complex) A set Σ of simplices is called a *triangulation*. A *simplicial complex*, often simply denoted as *complex*, is a triangulation Σ that verifies the following conditions:

1. if $\sigma \in \Sigma$, then any face of σ belongs to Σ;
2. if $\sigma, \tau \in \Sigma$, then either $\sigma \cap \tau = \emptyset$, or $\sigma \cap \tau$ is a face of both σ and τ.

A simplicial complex can be considered a *well-formed* triangulation. Such kind of triangulations are widely used in engineering analysis, e.g., in topography or in finite element methods.

Coding 3.2.3 (binomial numbers and simplex faces) There are

$$\sum_{k=0}^{n} \binom{n}{k} = 2^n$$

k-faces in a n-simplex. Testing this statement is simple and very elegant with Julia. We may verify, e.g., that `simplex(5)` faces are a set of 2^5 elements:

```
julia> [binomial(5,k) for k=0:5]'
1×6 adjoint(::Vector{Int64}) with eltype Int64:
 1  5  10  10  5  1

julia> [binomial(5,k) for k=0:5] |> sum == 2^5
true
```

Coding 3.2.4 A simple coding example produces the simplicial complex providing the triangulation of the boundary of a convex hull.

```
julia> p = rand(6,3)                    #=
6×3 Matrix{Float64}:
 0.456121  0.340689  0.523394
 0.670731  0.920846  0.810581
 0.511325  0.83709   0.765548
 0.27295   0.344676  0.246891
 0.155611  0.262588  0.372059
 0.997037  0.689132  0.594624         =#
```

The `p` matrix holds 6 random `3D` points. The `q` matrix receives an array of arrays:

```
julia> q = [p[k,:] for k=1:size(p,1)]              #=
6-element Vector{Vector{Float64}}:
 [0.4561213293752391, 0.34068894716553066, 0.5233939444809501]
 [0.6707310911896536, 0.9208461259235498, 0.8105811405026019]
 [0.5113250218604997, 0.8370897242704682, 0.7655476328700838]
 [0.2729499977250678, 0.3446764528053594, 0.24689130455109154]
 [0.15561059083290985, 0.26258788197490757, 0.37205895443742]
 [0.9970371401112009, 0.6891321979374976, 0.5946242627157257]
    =#
```

Their convex hull is computed by the `Plasm` function `CONVEXHULL` and stored as `Hpc` data structure:

```
julia> CONVEXHULL(q)                    #=
Hpc(MatrixNd(4), Geometry([[0.4561213293752391,
    0.34068894716553066, 0.5233939444809501],
    [0.6707310911896536, 0.9208461259235498,
    0.8105811405026019], [0.5113250218604997,
    0.8370897242704682, 0.7655476328700838],
    [0.2729499977250678, 0.3446764528053594,
    0.24689130455109154], [0.15561059083290985,
    0.26258788197490757, 0.37205895443742], [0.9970371401112009,
     0.6891321979374976, 0.5946242627157257]], hulls=[[1, 2, 3,
    4, 5, 6]]))               =#
```

And finally the dataset is transformed into a `Lar` data structure, which contains all the boundary polygons `:FV` and edges `:EV`.

```
julia> LAR(CONVEXHULL(q))                   #=
Lar(3, 3, 6, [0.27294999772507 0.67073109118965 …
    0.15561059083291 0.5113250218605; 0.34467645280536
    0.92084612592355 … 0.26258788197491 0.83708972427047;
    0.24689130455109 0.8105811405026 … 0.37205895443742
    0.76554763287008], Dict{Symbol, AbstractArray}(:CV => [[1,
    2, 3, 4, 5, 6]], :FV => [[1, 2, 3], [2, 3, 4], [1, 4, 5],
    [1, 3, 4], [1, 5, 6], [1, 2, 6], [4, 5, 6], [2, 4, 6]], :EV
    => [[2, 3], [1, 3], [1, 2], [3, 4], [2, 4], [1, 4], [4, 5],
    [1, 5], [5, 6], [1, 6], [2, 6], [4, 6]]))              =#
```

We remark that the `Hpc` describes only the *convex cells* of its dataset. Conversely, the `Lar` data structure gives explicitly all cells of any dimension. □

The *order* of a complex is the maximum order of its simplices. A complex Σ_d of order d is also called a *d-complex.* A d-complex is said to be *regular* or *pure* if each simplex is a face of a d-simplex. A regular d-complex is homogeneously d-dimensional.

The *combinatorial boundary* $\Sigma_{d-1} = \partial\sigma_d$ of a simplex σ_d is a simplicial complex consisting of all proper s-faces ($s < d$) of σ_d.

Two simplices σ and τ in a complex Σ are called *s-adjacent* if they have a common s-face. Hereafter, when we refer to adjacencies into a d-complex, we intend to refer to the maximum order adjacencies, i.e. to $(d-1)$-adjacencies. K_s ($s \leq d$) denotes the set of s-faces of Σ_d.

Simplex orientation

The ordering of 0-skeleton of a simplex implies an *orientation* of it.

Definition 3.12 (Ordering of simplex vertices) The simplex can be oriented according to the even or odd permutation class of its 0-skeleton. □

The two opposite orientation of a simplex will be denoted as $+\sigma$ and $-\sigma$.

Definition 3.13 (Coherent orientation) Two simplices are *coherently oriented* when their common faces have opposite orientation. A complex is *orientable* when all its simplices can be coherently oriented. □

It is assumed that:

1. the two orientations of a simplex represent its relative interior and exterior;
2. the two orientations of an orientable simplicial complex analogously represent the relative interior and exterior of the complex, respectively;
3. the boundary of a complex maintains the same orientation of the complex.

The volume associated with an orientation of a simplex (or complex) is positive, while the one associated with the opposite orientation has the same absolute value and opposite sign. It is assumed that the bounded object has positive volume. It is also assumed that either a minus sign or a multiplying factor -1 denote a complementation, i.e. an opposite orientation of the simplex, which can be explicitly obtained by swapping two vertices in its ordered 0-skeleton. For example:

$$+\sigma_3 = \langle v_0, v_1, v_2, v_3\rangle, \qquad -\sigma_3 = \langle v_1, v_0, v_2, v_3\rangle.$$

Definition 3.14 (Volume of a 3-simplex) The volume of a 3-simplex is a 2-cochain (see Section 3.3) from the cycle of its 2-faces (boundary triangles) to $\mathbb{R}$. By definition, a 2-cochain is a map from 2-chains to $\mathbb{R}$. □

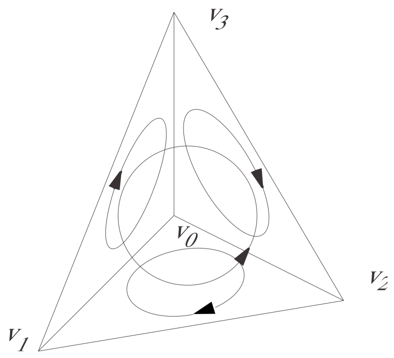


Fig. 3.2 Coherent orientation of the 2-faces of a 3-simplex

Facet extraction

The $(d-1)$-faces of a d-dimensional simplex or complex are also called *facets* since no ambiguity may arise. It is possible to see [?] what follows:

Theorem 3.1 (Whitney, 1957) *The oriented facets $\sigma_{d-1}^{(i)}$ $(0 \le i \le d)$ of the oriented d-simplex $\sigma_d = +\langle v_0, v_1, \ldots, v_d \rangle$ are obtained by removing the i-th vertex v_i from the 0-skeleton of σ_d:*

$$\sigma_{d-1}^{(i)} = (-1)^i (\sigma_d - \langle v_i \rangle), \qquad 0 \le i \le d. \tag{3.3}$$

The 0-skeleton of $\sigma_{d-1,(i)}$ is therefore obtained by removing the i-th vertex from the 0-skeleton of σ_d and either by swapping a pair of vertices or, better, by inverting the simplex sign, when i is odd.

Coding 3.2.5 (Facets of a d-simplex.) A julia function is given here to compute combinatorially all the oriented facets of a `simplices` arrays of d-simplexes. A precondition is that all `Vector{Int64}` in the input sequence have the same length.

```
function simplexfacets(simplices)
   @assert hcat(simplices...) isa Matrix
   out = Array{Int64,1}[]
    for simplex in simplices
        for v in simplex
            facet = setdiff(simplex,v)
            push!(out, facet)
        end
    end
    # remove duplicate facets
    return sort(collect(Set(out)))
end
```

If the `expr` after `@assert` is `false`, the program stops in `AssertionError: expr` □

Coding 3.2.6 (Execution example.) We can start from an array of simplices of the same length and iterate on their facets, to get a whole simplicial complex.

```
FV=[[123],[124],[134],[234]] #=
[[123],[124],[134],[234]] =#
EV = simplexfacets(FV)  #=
[[12],[13],[14],[23],[24],[34]] =#
VV = simplexFacets(EV)  #=
[[1], [2], [3], [4]]     =#
simplex(3,complex=true).C   #=
Dict(:c1v => EV, :c3v => [[1,2,3,4], :c0v => VV], :c2v => FV) =#
```

We can also generate directly the whole bounday complex of a multidimensional simplex

Coding 3.2.7 (Generation of boundary complex) We can olso directly generate the `Lar` boundary complex of a simplex of any degree:

```
simplex(4,complex=true).C           #=
Dict{Symbol, AbstractArray} with 5 entries:
  :C4V => [[1, 2, 3, 4, 5]]
  :C3V => [[1, 2, 3, 4], [1, 2, 3, 5], [1, 2, 4, 5], [1, 3, 4,
    5], [2, 3, 4, 5]]
  :C0V => [[1], [2], [3], [4], [5]]
  :C2V => [[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3,
    5], [1, 4, 5], [2…
  :C1V => [[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2,
    5], [3, 4], [3, …  =#
simplex(4,complex=true).V           #=
4×5 Matrix{Float64}:
 0.0  1.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  1.0               =#
```

Coding 3.2.8 (Simplex generation in `Hpc`) Convnersely, the `Plasm` data structure `Hpc` contains a single convex cell:

```
SIMPLEX(4)          #=
Hpc(MatrixNd(5), Geometry([[0.0, 0.0, 0.0, 0.0], [1.0, 0.0, 0.0,
     0.0], [0.0, 1.0, 0.0, 0.0], [0.0, 0.0, 1.0, 0.0], [0.0,
    0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4, 5]]))          =#
```

Definition 3.15 (Simplicial extrusion)

The prism over a simplex $\sigma_d = \langle v_0, \dots, v_d \rangle$, defined as the set $P_{d+1} := \sigma_d \times [a,b]$, with $[a,b] \subset \mathbb{E}$, will be called *simplicial* $(d+1)$*-prism*. An oriented complex which triangulates P_{d+1} can be defined combinatorially, by using a closed form formula for its K_{d+1} skeleton:

$$K_{d+1} = \{\sigma_{d+1,(i)} = (-1)^{id} \langle v_i^a, v_{i+1}^a, \dots, v_d^a, v_0^b, v_1^b, \dots, v_i^b \rangle, \quad 0 \le i \le d\}$$

where $v_i^a = (v_i, a)$ and $v_i^b = (v_i, b)$.

Closed formulas to triangulate the $(d+1)$-prism over a d-complex in a time linear with the size of the output, while computing also the d-adjacencies between the resulting $(d+1)$-simplices, can be found in [**?**, **?**]. □

Simplicial grids

In layout design, a grid system provides a framework of intersecting vertical and horizontal lines. Designers use this framework to place and align text, images, and other elements.

Analogously, a *simplicial grid*, is a geometric grid structure made by simplicial cells of same dimension aligned on a layout grid 1D, 2D, 3D, etc. They are mainly used for domain decomposition, where to map coordinate functions in order to create curved structures, i.e., proper curves, curved surfaces, and curved solids.

The whole curved complex is easily generated by mapping the vertices of a `Lar` grid.

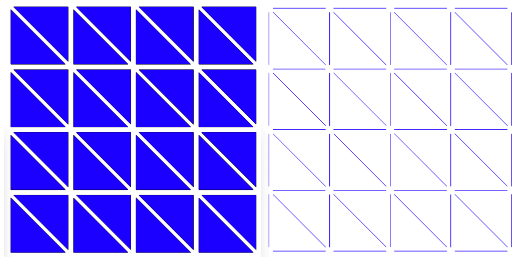


Fig. 3.3 The simplicial 2- and 1-complex (shown here slightly exploded) are generated by *extrudecomplex()* using the same pattern, both starting from *model1D*. The 1-complex is derived by the 2-complex using *simplexfacets()*.

Coding 3.2.9 (1D Extrusion)

```
V1 = [ 0.0 1.0 2.0 3.0 4.0 5.0 ]
EV = [[1, 2], [2, 3], [3, 4], [4, 5]]
```

```
mod1D = Lar(V1, Dict(:c1v => EV))
V2 = [mod1D.V; zeros(size(mod1D.V,2))']
GL.VIEW(GL.GLExplode(V2, map(x->[x],EV),1.1,1.1,1.1,2,1));
```

The mod2D and mod3D grids are generated by the following code.

Coding 3.2.10 (2D Extrusion)

```
mod2D = extrudecomplex(mod1D::Lar, [1, 1, 1, 1])
FV = collect(values(mod2D.C))[1]
EV = simplexfacets(FV)
GL.VIEW(GL.GLExplode(mod2D.V, map(x->[x],EV),1.2,1.2,1.2,4,1));
```

Coding 3.2.11 (3D Extrusion)

```
mod3D = extrudecomplex(mod2D::Lar, [1,-2,1])
CV = collect(values(mod3D.C))[1]
FV = simplexfacets(CV)
EV = simplexfacets(FV)
GL.VIEW(GL.GLExplode(mod3D.V,map(x->[x],EV),2,2,2,99,1));
```

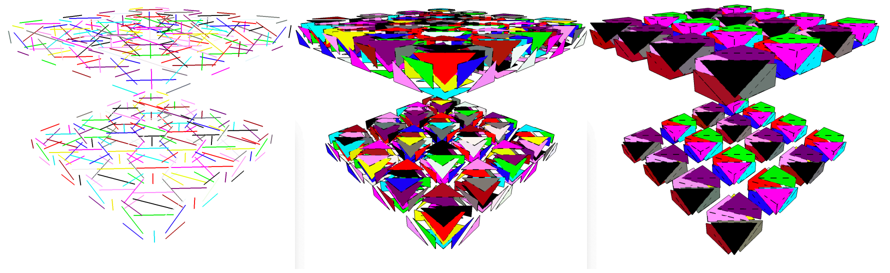


Fig. 3.4 Exploded simplicial grids of dimensions 1 (lines), 2 (surfaces), and 3 (solids). The simplicial 2-complex is generated by *simplexfacets()* from the 3-complex *extrudecomplex(simplex2D)*, with *simplex2D* in turn starting from *model1D*, i.e., from the expression generating the simplicial 1-complex .

3.2.2 Cubical complex and grid

We introduce now the definition and use of multidimensional *grids* of *cuboidal*, and the more general *Cartesian product* of cellular complexes. Such operators, depending on the dimension of their input, may generate either *full-dimensional* (i.e. solid) output complexes, or *lower-dimensional* complexes of dimension d embedded in Euclidean n-space, with $d \leq n$.

Regular grid

Regular cubical grids appear in finite element analysis, finite volume methods, finite difference methods, and in general for discretization of parameter spaces.

Definition 3.16 (Regular cuboidal grid) A regular grid is a tessellation of n-dimensional Euclidean space by congruent parallelotopes (e.g. bricks). Its opposite is irregular grid. We prefer to call these objects "cuboidal" complexes, in order to remark their multidimensional cheracter in `Plasm`. □

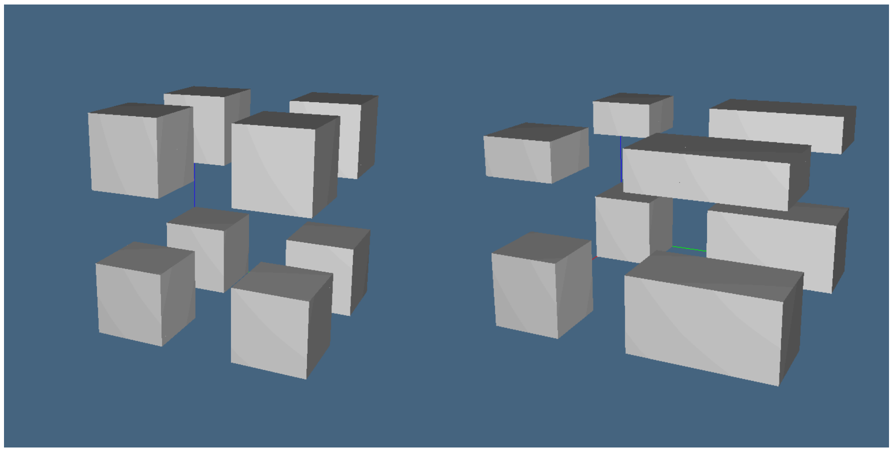


Fig. 3.5 Unconnected cubical grids: (a) regular cell complex; (b) irregular cell complex. See Script 3.2.12.

E.g., just think to a mesh of 3D cubes in three-dimensional space for the first case, and to the (non-manifold) framework of skeletal polygons of such cubic cells for the second case.

In particular, both n-dimensional *solid grids* of (hyper)-cuboidal cells and their d-dimensional *skeletons* $(0 \leq d \leq n)$, embedded in $\mathbb{E}^n$, are generated by assembling the cells produced by a product of n either 0D or 1D cellular complexes, that in such lowest dimensions coincide with simplicial complexes.

In `Plasm` the cuboidal grids are not necessarily regular, and can be even unconnected.

Coding 3.2.12 (3D Cell Complex) Two simple examples of 3D cuboidal grids are shown in Figure 3.5. Each `GRID` instance generates a 1D cellular complex, and two Cartesian product, denoted in `Plasm` by the `*` infix symbol, produce the 3D cell complex.

```
VIEW(GRID([1,-1,1]) * GRID([1,-1,1]) * GRID([1,-1,1]))
VIEW(GRID([1,-2,1]) * GRID([1,-1,2]) * GRID([1,-1,0.5]))
```

It is easy to believe that negative parameters denote empty space, while the positive ones denote full space. See Figure 3.5. □

3.2.3 Polyhedral complex

Topological product

In this section, we discuss a solid modeling operation highly useful to generate 3D solids from 2D surfaces, as well as 2-complexes from 1-complexes, and to produce their skeletons taking advantage of one or more 0-complexes.

Definition 3.17 (Topological product) A product space is the *Cartesian product* of a family of topological spaces equipped with a natural topology.□

The natural topology on a subset of a topological space, in our case $\mathbb{E}^d$, is the relative topology (or subspace topology). The product of two cell complexes can be made into a cell complex as we show in the following.

Definition 3.18 (Product of cell complexes) If X and Y are m- and n-complexes in $\mathbb{E}^m$ and $\mathbb{E}^n$, respectively, $X \times Y \subset \mathbb{E}^{m+n}$ is a cell complex in which each cell is a product of a cell in X and a cell in Y, endowed with the relative topology in $\mathbb{E}^{m+n}$. □

Cell complex product

Definition 3.19 (larprod) The `Plasm` `POWER` function, also denoted as infix `*`, takes as input a pair of `Plasm` models of `Hpc` type, and returns the value `Hpc` of their *Cartesian product*. □

The operation is associative, hence no parenthesis are needed, as you can see in Coding 3.2.12.

3.3 Chain complex

This section deals with algebraic topological tools to compute boundaries and adjacencies of *chains* of cells, and to generate the discrete solid primitives needed by a computational modeler. Basic geometric and algebraic topology provide the mathematical concepts needed to compute and explore the cells of the space partition induced by a set of geometric objects, as well as the related incidence/neighborhood relations.

To work over huge geometric datasets we need explicit expression of topological relations using proper and efficient computational structures. The purpose of this chapter is to elucidate our best methods for multidimensional solid modeling computing. The geometric discussion is restricted to piecewise-linear topology and to space dimensions less or equal to three.

Some words about notations: greek letters are used for the *cells* of a space partition, and roman letters for *chains* of cells, coded in coordinates as either signed integers or sparse arrays of signed integers.

3.3.1 Linear chain spaces

In mathematics, a graded vector space is a vector space that has the extra structure of a gradation, which is a decomposition of the space into a direct sum of vector subspaces, generally indexed by the integers.

Definition 3.20 (Graded vector space) A *graded vector space* is a vector space V expressed as a direct sum of subspaces V_p indexed by integers in $[0,d] := \{p \in \mathbb{N} \ : \ 0 \leq p \leq d\}$:

$$V = \oplus_{p=0}^{d} V_p. \tag{3.4}$$

Definition 3.21 (Graded linear maps) A linear map $f : V \to W$ between graded vector spaces is a *graded map* of degree k if $f(V_p) \subset W_{p+k}$ for each p. □

Let be given a cellular complex X. We are interested in working with subsets of cells with the *same* dimension, and with their maps 1-graded, which completely specify the X topology.

Definition 3.22 (p-chain) A *p-chain* can be seen, with some abuse of language, as *any subset* of p-cells in a cellular complex X. □

Definition 3.23 (Unit p-chain) In this sense we write $U_p = \Lambda_p - \Lambda_{p-1}$ for the set of *unit* (or *elementary*) p-chains ($0 \leq p \leq d$), where Λ_p is the p-skeleton of X and $\Lambda_{-1} = \emptyset$. □

Definition 3.24 (Linear p-ppace) $C_p = \mathscr{P}(U_p)$ is the space of p-chains, where $\mathscr{P}$ is the power set. □

The set $C = \oplus\, C_p$, direct sum of chain spaces, can be given the structure of a graded vector space (see 3.20 and [**?**]) by defining sums of chains with the same dimension, and products times scalars in a field, with the usual properties.

Definition 3.25 (Chain bases) As a linear space, each C_p contains a set of irreducible generators. The natural *basis* $U_p \subset C_p$ is the set of *independent* (or *elementary*) chains $u_p \in C_p$, given by singleton elements. Consequently, every chain $c \in C_p$ can be written as a linear combination of this basis with field elements, and is uniquely generated. □

Once the basis is fixed, *i.e.*, U_p is ordered, the unique unsigned coordinate representation of each $\{\lambda_k\} =: u_k \in C_p$ is a binary array with one non-zero element in position k, and all other elements 0. The ordered sequence of scalars may be drawn either from $\{0, 1\}$ (unsigned representation) or from $\{0, 1, -1\}$ (oriented representation). With abuse of language, we often call p-cells the elements of U_p.

Definition 3.26 (Chain complex) A *chain complex* is a graded vector space V provided with a graded linear map $\partial : V \to V$ of degree -1 called *boundary operator*, which satisfies $\partial^2 = 0$. □

In other words, a chain complex is a sequence of vector spaces C_p and linear maps $\partial_p : C_p \to C_{p-1}$, such that $\partial_{p-1} \circ \partial_p = 0$. The notation $C_\bullet$ is used in this paper for the chain complex over the binary field $\{0, 1\}$, and $C_\bullet^\circlearrowleft$ for the oriented chain complex over the ternary field $\{0, 1, -1\}$, used to get oriented boundaries.

Definition 3.27 (Cochain complex) A *cochain complex* is a graded vector space V furnished with a graded linear map $\delta : V \to V$ of degree $+1$ called *coboundary operator*, which satisfies $\delta^2 = 0$. □

That is to say, a cochain complex is a sequence of vector spaces C^p and linear maps $\delta^p : C^p \to C^{p+1}$, such that $\delta^{p+1} \circ \delta^p = 0$.

3.3.2 Linear chain operators

In the remainder of this book we identify chains and cochains, as in [**?**]. In this way we express to be only interested to combinatorial topology aspects of cell complexes, and not in differential ones.

A standard basis, also called a natural basis, is a special orthonormal vector basis in which each basis vector has a single nonzero entry with value 1 [**?**]. Being only interested in topological properties, we have identified elementwise the natural bases of the corresponding chain and cochain spaces.

As a consequence, the boundary and coboundary matrices are related by transposition, and given the coordinate vector of a p-cell, provide both the incident $p-1$ and $p+1$ cells, respectively.

Given a collection S of geometric objects[3], we shall compute the topology of their space arrangement $\mathcal{A}(S)$ as a *chain complex*, *i.e.*, as a sequence of linear spaces C_p of chains and linear boundary/coboundary maps ∂_p and $\delta_p = \partial_{p+1}^\top$ between them:

$$C_\bullet = (C_p, \partial_p) := C_3 \underset{\partial_3}{\overset{\delta_2}{\leftrightarrows}} C_2 \underset{\partial_2}{\overset{\delta_1}{\leftrightarrows}} C_1 \underset{\partial_1}{\overset{\delta_0}{\leftrightarrows}} C_0.$$

Operator matrices

The matrices of boundary and coboundary operators (their transpose) are very sparse, with sparsity growing linearly with the number n of rows (sparse columns in Julia). With common data structures [**?**] for sparse matrices, the storage cost $O(n)$ is linear with the number of cells, with $O(1)$ small cost per cell that depends on the storage scheme.

According to what discussed in Section 3.3.2, we may identify the U_0 basis of 0-chains with the set `V` of vertices (0-cells) of a cellular complex, and the U_1, U_2 bases of 1- and 2-chains with the set `E` of edges (1-cells) and `F` of faces (2-cells), respectively. Analogously, we use the symbol `C` for the set of 3-cells.

Therefore, we adopt a pair of characters from {`V`, `E`, `F`, `C`}, for instance `EF`, to denote the matrix of mapping `F` → `E` from 2-chains in `F` to 1-chains in `E`, in mathematical terms $\partial_2 : C_2 \to C_1$ (see Section 3.1). Hence, for sake of readiness and simplicity we will often use the following notations:

$\delta_2 : C_2 \to C_3 \equiv$ `CF: F → C` $\delta_1 : C_1 \to C_2 \equiv$ `FE: E → F`
$\delta_0 : C_0 \to C_1 \equiv$ `EV: V → E`

$\partial_3 : C_3 \to C_2 \equiv$ `FC: C → F` $\partial_2 : C_2 \to C_1 \equiv$ `EF: F → E`
$\partial_1 : C_1 \to C_0 \equiv$ `VE: E → V`

Remark 3.4 (Matrix and vector formats) Actual content of vectors `V,E,F,C` is the coordinate representation of a chain (subset) of topological entities, hence the characteristic function of such subset of cells (binary vector). □

Remark 3.5 (Linear operator vs matrix) While linear operators work between two chain spaces, the actual application of their matrix produces a range vector from a domain vector (both in coordinates). □

[3] Examples include, but are not limited to: line segments, quads, triangles, polygons, meshes, pixels, voxels, volume images, B-reps, *etc.* In mathematical terms, a geometric object is a topological space embedded in some $\mathbb{E}^d$ [**?**].

Matrix-Vector Products as Linear Combinations

Left-multiply a matrix A by a *row vector* x, i.e. xA, gives a linear combination of A rows with x elements as scalars, and produces a row vector. Analogously, right-multiply a matrix A by a *column vector* x, i.e. Ax, gives a linear combination of A columns, and produces a column vector.

Coding 3.3.1 (Construction of operator matrix) $[\partial_2]$. With some abuse of language (removed after Section **??**), we build the sparse matrix of `EF ≡ ∂₂: F → E`.

```
∂₂ = K(EV) * K(FV)' .÷ 2  #=
12×8 SparseMatrixCSC{Int64, Int64} with 24 stored entries:
 ⋅  ⋅  ⋅  ⋅  ⋅  1  ⋅  1
 ⋅  ⋅  1  ⋅  ⋅  1  ⋅  ⋅
 ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅  1
 ⋅  ⋅  1  1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1  ⋅  1  ⋅
 1  ⋅  ⋅  ⋅  1  ⋅  ⋅  ⋅
 ⋅  1  ⋅  ⋅  ⋅  ⋅  1  ⋅
 1  1  ⋅  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  1  1  ⋅  ⋅
 ⋅  ⋅  ⋅  ⋅  ⋅  ⋅  1  1
 1  ⋅  1  ⋅  ⋅  ⋅  ⋅  ⋅
 ⋅  1  ⋅  1  ⋅  ⋅  ⋅  ⋅   =#
```

It may be interesting to make some notation about this matrix. Remember that, by right multiplication, any mapping matrix sends the space of columns (2-chains: triangles here) to the space of rows (1-chains: edges here). With any simplicial complex we have two 1s on each row and three 1s on each column. They characterize the two faces incident on the edge, and the three edges incident on the face, respectively. □

Coding 3.3.2 (How to use the linear operators.) A boundary operator, which is a linear map, sends every vector in domain space (the span of the matrix columns) to its image vector in target space (the span of the matrix rows).

```
face = zeros(length(FV)); face[4] = 1
(∂₂ * face)'  #=
1×12 adjoint(::Vector{Float64}) with eltype Float64:
 0.0  0.0  1.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0   =#
```

In particular, the matrix of a linear operator—coordinate representation of it with respect to the chosen bases—send the coordinate representation of a domain vector to its image in target space. Both are binary vectors, as will be clear after Section **??**. E.g., `face[4]` has boundary edges 3, 4, and 12 (look for `1`s positions in row 5 above). □

Coding 3.3.3 (Cell with a hole) Figure 3.6 shows an example of 2D cellular complex $X = X_2$, comprised of 8 unit 0-chains (0-cells) u_0^h, 8 unit 1-chains (1-cells) u_1^k, and 2 unit 2-chains (2-cells) u_2^j.

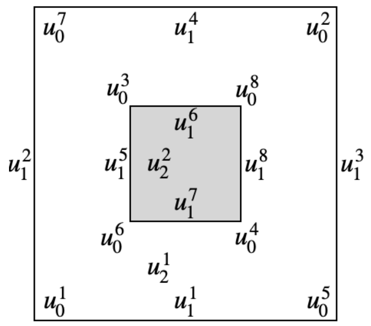


Fig. 3.6 Cellular 2-complex with two 2-cells, eight 1-cells, and eight 0-cells.

A user-readable representation of the geometric complex (X_2, ν) is given below. `V` is the array of vertices, that provides the embedding map $C_0 \to \mathbb{E}^2$, implemented as array $\nu : \mathbb{N} \to \mathbb{R}^2$. `EV` and `FV` respectively provide the *canonical* (sorted) LAR of 1-cells and 2-cells as lists of lists of 0-cells indices. These can be interpreted as user-readable CSR (Compressed Sparse Row) characteristic matrices M_1 and M_2 of the 0-generators of 1-cells and 2-cells, respectively, according to [**?**].

```
V = [[0.,0.],[3,3],[1,2],[2,1],[3,0],[1,1],[0,3],[2,2]]
FV = [[1,2,3,4,5,6,7,8],[3,4,6,8]]
EV = [[1,5],[1,7],[2,5],[2,7],[3,6],[3,8],[4,6],[4,8]]
```

The unsigned matrix of the boundary operator $\partial_2 : C_2 \to C_1$, computed by filtering elements of value 2 in the matrix $M_1 M_2^t$, is:

$$[\partial_2] = \text{filter}(M_1 M_2^t, 2) = \text{filter}\left(\begin{pmatrix} 1&0&0&0&1&0&0&0 \\ 1&0&0&0&0&0&1&0 \\ 0&1&0&0&1&0&0&0 \\ 0&1&0&0&0&0&1&0 \\ 0&0&1&0&0&1&0&0 \\ 0&0&1&0&0&0&0&1 \\ 0&0&0&1&0&1&0&0 \\ 0&0&0&1&0&0&0&1 \end{pmatrix} \begin{pmatrix} 1&0 \\ 1&0 \\ 1&1 \\ 1&1 \\ 1&0 \\ 1&1 \\ 1&0 \\ 1&1 \end{pmatrix}, 2 \right) = \begin{pmatrix} 1&0 \\ 1&0 \\ 1&0 \\ 1&0 \\ 1&1 \\ 1&1 \\ 1&1 \\ 1&1 \end{pmatrix}$$

where the first column represents the non-convex 2-cell with the hole, and the second column represents the convex cell within the hole. The reader may easily check that the four ones in positions from fifth to eighth in the second column of $[\partial_2]$ correspond to the last four unit 1-chains in `EV` array. By multiplication (mod 2) of $[\partial_2]$ times the coordinate representation $[c]$ of the 2-complex in Figure 3.6, *i.e.*, times the *total* 2-chain $c = u_2^1 + u_2^2 = \begin{pmatrix} 1 & 1 \end{pmatrix}^t$, we get the coordinate representation

$$[\partial_2][c] = \left(1\ 1\ 1\ 1\ 0\ 0\ 0\ 0 \right)^t$$

of the 1-boundary of c, *i.e.*, the cycle $u_1^1 + u_1^2 + u_1^3 + u_1^4$ made by the first four 1-cells in `EV`. □

3.4 Cochain integration (surface, volume, inertia)

Coding 3.4.1 (Volume of a cube from boundary triangles) . The variable `obj` first contains the memory pointer to an object of type `Hpc`, then to an object of type `Lar`. Here, `obj.C` gives the dictionary `C` of its cell complex, where `C[:FV]` denotes the array of boundary faces (triangles).

```
julia> obj = SIMPLEX(3)     #=
Hpc(MatrixNd(4), Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
    [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]], hulls=[[1, 2, 3, 4]])) =#


julia> obj = LAR(SIMPLEX(3))    #=
Lar(3, 3, 4, [0.0 1.0 0.0 0.0; 1.0 0.0 0.0 0.0; 0.0 0.0 0.0
    1.0], Dict{Symbol, AbstractArray}(:CV => [[1, 2, 3, 4]], :FV
     => [[1, 2, 3], [2, 3, 4], [1, 3, 4], [1, 2, 4]], :EV =>
    [[2, 3], [1, 3], [1, 2], [3, 4], [2, 4], [1, 4]])) =#

FV = obj.C[:FV]  #=
4-element Vector{Vector{Int64}}:
 [1, 2, 3]
 [1, 2, 4]
 [1, 3, 4]
 [2, 3, 4]   =#
P = (obj.V, FV); VOLUME(P)  #=
0.16666666666666666   =#
volume(P) * 6   #=
1.0   =#
```

Part II
Dimension-independent Modeling

Chapter 4
Geometric models

Julia `Plasm` is the best choice to develop geometric models for Building Information Modeling (BIM) and Computer-Aided Design (CAD). We ported the functional language to Julia for better supporting design, model generation, and visualization of geometric objects. Geometric models specify the physical appearance of Architecture, Engineering, and Construction products at any scale, from structural and envelope components to whole buildings and built environments, and are used for design, tender, contract, and collaboration. In this chapter we introduce the great expressive power of `Plasm` parametric functions and the simple methods of parametric assemblies, where objects itself can be used as actual parameters. We show also that `Plasm` offers a very general mechanism (Julia dictionaries) to characterize and export models with colors, textures, materials, and so on. `Plasm` can be even embedded in the Jupiter platform to document the design choices step-by-step.

4.1 Introduction to geometric objects

4.2 Plasm parametric primitives

4.3 Hierarchical assembly of geometric objects

4.4 Attach properties to geometry

4.5 Design documentation (Jupyter notebooks)

4.6 Export geometry

Chapter 5
Symbolic modeling with Julia Plasm

Symbolic modeling is a semantic approach to knowledge representation and processing. The symbolic approach to knowledge representation and processing uses names to define the meaning of represented knowledge explicitly. The geometric knowledge is described here by Julia's names, which are given to functionals, functions, formal and actual parameters, and finally to objects, fields, classes, attributes, methods, relations, etc. In this chapter, we give many examples of high-level Plasm programming, from topological, linear, and affine operators, to topological mapping of complexes and grids to generate linearized approximation of curved manifold of intrinsic dimensions 1, 2, and 3. i.e., depending on such number of parameters; say, curves, surfaces, thin, and bulk solids.

5.1 Primitive generators

5.2 Plasm topological operators

5.3 Linear and affine operators

5.4 Manifold mapping

5.5 Predefined Plasm functions

5.6 Curve, surface, and solid methods

Chapter 6
Product assembly structure

Hierarchical models of assemblies are generated by aggregating subassemblies, each defined in its local coordinate system and relocated by affine transformations. This operation may be repeated hierarchically, with some subassemblies defined by aggregating simpler parts, and so on, until we get a set of elementary components that cannot be further disaggregated. Two main advantages can be found in hierarchical modeling. Each elementary part and each assembly, at every hierarchical level, are defined independently from each other, using local coordinate frames suitably chosen to make their definition easier. Furthermore, only one copy of each component is used, and it can be instanced in different locations and orientations, depending on how many times it is needed. In this chapter, after an overview of solid modeling data structures, the models of the `Plasm` dataset are discussed.

6.1 Hierarchical ssembly definition

6.2 Data structures in solid modeling

6.3 Structure in PHIGS and Plasm

6.4 Julia Plasm data structures

6.4.1 Hierarchical Polyhedral Complex (HPC)

6.4.2 Linear Algebraic Representation (LAR)

6.4.3 Geometric DataSet (GEO)

Chapter 7
Space arrangements

The chapter introduces computational topology algorithms to discover the two-dimensional (2D)/3D space partition induced by a collection of geometric objects of dimension (1D)/2D, respectively. The data structures needed for such computational program are sparse arrays and their standard algebraic operations. In this chapter, we introduce a novel approach to solid modeling based on piecewise-linear algebraic topology, that allows to treat rather general cellular complexes, with cells homeomorphic to polyhedra, i.e., to triangulable spaces, and hence possibly non-convex and multiply connected. The notions we deal with include geometric complexes, linear spaces of chains and cochains, the chain complex of linear operators between pairs of spaces, and their compositions. The discussion is restricted to piecewise-linear topology and to space dimensions less or equal to three.

7.1 Space partition and enumeration

7.2 Cellular and boundary models

7.3 Arrangements and Lattices

7.4 2D and 3D Examples

Chapter 8
Boolean solid algebras

Engineers conceived Solid Modeling as a rigorous and universal language for geometry-based engineering. Mathematically, all solid models are computer representations of elements of some algebraic systems that are constructed and maintained by algorithms corresponding to the operations in such algebra. This chapter, in particular, focuses on the algebraic relationships between CSG (Constructive Solid Geometry) and boundary representations of PL polyhedra. We show that the (regularized) arrangement of a given set of spatially instanced primitive shapes is isomorphic to the finite Boolean algebra of regularized sets containing all possible CSG representations with the same primitives. In particular, we show that the boundary evaluation of any CSG representation with a finite number of primitives reduces algebraically to a composition of Boolean operations on bit strings, sparse matrix multiplication, and point membership tests, and show significant examples.

8.1 Constructive Solid Geometry (CSG)

8.2 Atoms and Generators

8.3 Finite Boolean Algebras

8.4 Computational Pipeline

Part III
Polyhedral Modeling in AEC

Chapter 9
Building Information Modeling (BIM)

The construction industry is nowadays undergoing its very own digital revolution, helped by its unique way of working with BIM systems whose elements are the digital prototypes of physical elements like columns, windows, walls, doors, and stairs. In this chapter, we outline the conceptual foundations and the down of the BIM attitude of the construction industry, discuss the birth of concepts like building objects and the standardized taxonomy of building systems as set of subsystems devoted to the fulfillment of physical requirements of space with suitable performance levels. In particular, we show by example how Julia `Plasm`, relying on its understand of topology, may automatically transform the preliminary shape design of spaces into the generic geometry of building subsystems (skeleton, envelope, internal partitions, vertical communications) and elements.

9.1 BIM history (Chuck Eastman, ...)

9.2 Building taxonomy (UNI 9838)

9.3 Building envelope

9.4 Building skeleton

9.5 Construction Process Modeling

Chapter 10
Industry Foundation Classes (IFC)

The Industry Foundation Classes (IFC) are an open international standard for sharing Building Information Model (BIM) data. IFC is a non-proprietary file format used to exchange building information models. IFC files can share data between different software applications, making it possible for multiple stakeholders to collaborate on BIM projects. Of course, we are mostly interested in the IFC transport of shape information. In this chapter, we, therefore, investigate the Julia `Plasm` ability to implement in a few rows of parametric code the classes and attributes of the `IfcShapeRepresentation` entity for the various types of shape representation, including PointCloud, Curve, Surface, GeometricSet, GeometricCurveSet, Annotation2D, SurfaceModel, Tessellation, SolidModel, SweptSolid, AdvancedSweptSolid, Brep, CSG, Clipping, and BoundingBox.

10.1 Simple introduction to IFC

10.2 Data scheme for BIM collaboration

10.3 IfcShapeRepresentation (IFC 4.3.x: 8.18.3.14)

10.3.1 Representation identifiers

10.3.2 Representation types

10.3.3 Representation Examples

10.4 Plasm parametric programming to IFC

Part IV
Geometry from Point Cloud